

Computational Electrodynamics with Chombo

Part III

Zdzislaw Meglicki

Indiana University

A Work in Progress Report

Prepared for the Argonne National Laboratory

December 8, 2004


Document version \$Id: Metal.tex,v 1.15 2004/12/08 17:50:16 meglicki Exp \$

Abstract

This report discusses implementation of a multilevel adaptive mesh refinement (AMR) finite-difference time-domain (FDTD) transverse electric (TE) code that simulates scattering of a Gaussian pulse or a harmonic wave on various metal structures.

Perfectly matched layer (PML) absorbing boundary conditions (ABC) are provided to eliminate unwanted reflections off the domain boundary. The Gaussian pulse or a harmonic wave is injected into and then extracted from the total field region, within which the multilayer AMR computation unfolds. There is no limit, other than that imposed by available computer resources, on the number of AMR levels.

Gnuplot and HDF5 animations generated by the program illustrate propagation of electric and magnetic fields at all AMR levels, as well as AMR cell tagging and evolution of the fine level grid structure.

Open issues are flagged with the  sign on the margins. They are discussed, wherever there is anything constructive to say about them and related further development work is suggested.

Contents

1	The Drude Model	3
1.1	Maxwell Equations	4
1.2	Elimination of ϵ_0 and μ_0	7
1.3	Extracting \mathbf{E} from \mathbf{D}	8
1.3.1	The Double Current Method	9
1.3.2	The Single Current Method	9
1.3.3	The Z-transform Method	10
1.3.4	The ADE Method	12
1.3.5	Code Implementation	13
1.3.6	Summary	16
1.4	Medium Parameters in Natural Units	16
2	PML ABCs	18
3	Injection and Extraction of the Incident Field	21
3.1	Incident Field Models	25
4	Medium Distributions	26
5	Fortran Mainlines	28
5.1	Fortran Input	29
6	Chombo Mainlines	30
6.1	Chombo Input	32
7	Chombo Output	34
7.1	HDF5 Output	34
7.2	Gnuplot Output	35
8	Tests and Experiments	36
8.1	Plane Wave/Pulse propagation	36
8.2	Fortran versus Chombo	37
8.3	Single Level versus Multi-Level: Quenching of Spikes	41
9	What Next	46
9.1	Parallelization	46
9.2	Gnuplot Graphics	47
9.3	Production 2D Code	47
9.4	3D code	47
	References	48
	Index	49

1 The Drude Model

Here I merely quote various formulas of the Drude model without discussing where they come from. A good source in this case are Mike Nielsen's lecture notes, which were put on his private "blog", [10]. Just about all this stuff is covered in Feynman, [4], too, but it's not referred to as a "Drude Model". Also, Feynman scatters this material over various chapters and volumes even – so it's not all just in one place under the heading "Theory of Conduction in Metals".

As Michael Nielsen points out, quoting from "Solid State Physics" by Ashcroft and Mermins, [1], the Drude model of metals breaks in the quantum domain and even its various high temperature predictions do not agree with experiments very well, so that quantum theory has to be invoked in order to get correct expressions for conductivity. But the general form that the Drude model assumes within the framework of Maxwell equations works very well, as long as we use phenomenological expressions for plasma frequency, ω_p , and for the relaxation time, τ .

On the other hand I strongly suspect that the model may break in the nano-physics regime. In order to build a plasma mode that quenches an electromagnetic wave penetrating a nano-grain a certain amount of space is needed. If the grain is smaller, there may not be enough space in it for the mode to form and so the grain may prove more transparent than the Drude model would predict. But then quantum effects may produce surface plasmons and so the grain may actually end up being less transparent than the Drude model would predict.

Whichever is the case, we may end up observing a discrepancy between computational predictions derived from the classical Drude model alone and laboratory experiments.

Anyhow, here's the model.

- N is the number of valence electrons per unit volume.
- Z is the number of valence electrons per atom.
- N_A is the Avogadro number, i.e., the number of atoms per mol.
- ρ_m is the mass density of metal.
- A is the atomic weight of metal

$$N = Z \times N_A \times \rho_m / A$$

- \mathbf{j} is the current density.
- $\langle \mathbf{v} \rangle$ is the average velocity of electrons.
- τ is the collision time.
- σ is the conductivity.

$$\mathbf{j} = -Nq_e\langle \mathbf{v} \rangle$$

also

$$\langle \mathbf{v} \rangle = -q_e \mathbf{E} \tau / m_e$$

hence

$$\mathbf{j} = \frac{Nq_e^2\tau}{m_e} \mathbf{E}$$

and

$$\sigma = \frac{Nq_e^2\tau}{m_e}$$

- $\langle \mathbf{p} \rangle$ is the average electron momentum.
- \mathbf{F} is the force acting on electrons.

$$\frac{d\langle \mathbf{p} \rangle}{dt} = -\frac{\langle \mathbf{p} \rangle}{\tau} + \mathbf{F}$$

- ω frequency
- t time

Suppose

$$\mathbf{E}(t) = \mathbf{E}_0 e^{i\omega t}$$

Assume

$$\langle \mathbf{p} \rangle = \langle \mathbf{p} \rangle_0 e^{i\omega t}$$

then

$$i\omega \langle \mathbf{p} \rangle_0 = -\frac{\langle \mathbf{p} \rangle_0}{\tau} - q_e \mathbf{E}_0$$

hence

$$\langle \mathbf{p} \rangle_0 = \frac{-q_e \mathbf{E}_0}{i\omega + 1/\tau}$$

Since

$$\mathbf{j} = -Nq_e \langle \mathbf{v} \rangle$$

and

$$\langle \mathbf{v} \rangle = \langle \mathbf{p} \rangle / m_e$$

we get

$$\mathbf{j}_0 = -Nq_e \langle \mathbf{v} \rangle_0 = -Nq_e \langle \mathbf{p} \rangle_0 / m_e = \frac{-Nq_e}{m_e} \frac{-q_e \mathbf{E}_0}{i\omega + 1/\tau} = \frac{Nq_e^2 / m_e}{i\omega + 1/\tau} \mathbf{E}_0$$

where

$$\sigma = \frac{Nq_e^2 / m_e}{i\omega + 1/\tau}$$

1.1 Maxwell Equations

- \mathbf{M} is the magnetic current.
- $\mathbf{D} = \epsilon \mathbf{E}$
- $\mathbf{B} = \mu \mathbf{H}$
- $\epsilon = \epsilon_0 \kappa = \epsilon_0 (1 + \chi)$
- χ is the electric susceptibility.

$$\begin{aligned} \frac{\partial \mathbf{B}}{\partial t} &= -\nabla \times \mathbf{E} - \mathbf{M} \\ \frac{\partial \mathbf{D}}{\partial t} &= \nabla \times \mathbf{H} - \mathbf{J} \\ \nabla \cdot \mathbf{D} &= 0 \quad \text{in absence of free charges} \\ \nabla \cdot \mathbf{B} &= 0 \end{aligned}$$

Consider the $\nabla \times \mathbf{H}$ equation for the Drude model:

$$\nabla \times \mathbf{H} = \epsilon_0 \frac{\partial \mathbf{E}}{\partial t} + \sigma \mathbf{E}$$

Assuming again that $\mathbf{E} = \mathbf{E}_0 e^{i\omega t}$

$$\nabla \times \mathbf{H} = \epsilon_0 \frac{\partial}{\partial t} \left(\mathbf{E} + \frac{1}{i\omega} \frac{\sigma}{\epsilon_0} \mathbf{E} \right) = \frac{\partial}{\partial t} \epsilon_0 \left(1 + \frac{\sigma/\epsilon_0}{i\omega} \right) \mathbf{E}$$

hence we can rewrite this as

$$\nabla \times \mathbf{H} = \frac{\partial \mathbf{D}}{\partial t}$$

where

$$\begin{aligned} \mathbf{D} &= \epsilon_0 \left(1 + \frac{\sigma/\epsilon_0}{i\omega} \right) \mathbf{E} \\ &= \epsilon_0 \left(1 + \frac{Nq_e^2/(\epsilon_0 m_e)}{i\omega(i\omega + 1/\tau)} \right) \mathbf{E} \\ &= \epsilon_0 \left(1 + \frac{Nq_e^2/(\epsilon_0 m_e)}{-\omega^2 + i\omega/\tau} \right) \mathbf{E} \end{aligned}$$

But

$$1 + \frac{Nq_e^2}{\epsilon_0 m_e} \frac{1}{-\omega^2 + i\omega/\tau} = n^2$$

where

- n is the refraction index.
- $Nq_e^2/(\epsilon_0 m_e) = \omega_p^2$ is the plasma frequency
- and the above equation is the same as Feynman 32.38 (vol. II) [4] arrived at from the other side, i.e., by considering χ for plasma and then $\epsilon = \epsilon_0(1 + \chi)$.

So we can look at the plasma currents either as a contribution to $\mathbf{D} = \epsilon_0(1 + \chi)\mathbf{E}$ or we can look at them explicitly in terms of $\mathbf{j} = \sigma \mathbf{E}$ with the same effect in terms of Maxwell equations.

Using ω_p^2 in place of $Nq_e^2/(\epsilon_0 m_e)$ we get:

$$\mathbf{D} = \epsilon_0 \left(1 + \frac{\omega_p^2}{-\omega^2 + i\omega/\tau} \right) \mathbf{E}$$

Now, this equation holds *only* when both \mathbf{D} and \mathbf{E} vibrate with the same frequency ω , i.e., strictly speaking we should write here:

$$\hat{\mathbf{D}}(\omega) e^{i\omega t} = \epsilon_0 \left(1 + \frac{\omega_p^2}{-\omega^2 + i\omega/\tau} \right) \hat{\mathbf{E}}(\omega) e^{i\omega t}$$

with $\mathbf{D}(t)$ and $\hat{\mathbf{E}}(\omega)$ connected by:

$$\mathbf{D}(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{\mathbf{D}}(\omega) e^{i\omega t} d\omega = \frac{1}{2\pi} \int_{-\infty}^{\infty} \epsilon_0 \left(1 + \frac{\omega_p^2}{-\omega^2 + i\omega/\tau} \right) \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega$$

and

$$\mathbf{E}(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega$$

I have multiplied the integrals by $1/(2\pi)$ in order to make them into full inverse Fourier transforms. The transformation in the other direction is without the $1/(2\pi)$ factor:

$$\begin{aligned} \hat{\mathbf{D}}(\omega) &= \int_{-\infty}^{\infty} \mathbf{D}(t) e^{-i\omega t} dt \\ \hat{\mathbf{E}}(\omega) &= \int_{-\infty}^{\infty} \mathbf{E}(t) e^{-i\omega t} dt \end{aligned}$$

The first term under the integral simply becomes $\mathbf{E}(t)$ so:

$$\mathbf{D}(t) = \epsilon_0 \mathbf{E}(t) + \frac{\epsilon_0}{2\pi} \int_{-\infty}^{\infty} \frac{\omega_p^2}{-\omega^2 + i\omega/\tau} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega$$

The fraction $\omega_p^2/(-\omega^2 + i\omega/\tau)$ can be split into a sum of two simple fractions:

$$\frac{\omega_p^2}{-\omega^2 + i\omega/\tau} = \frac{\omega_p^2 \tau}{i\omega} - \frac{\omega_p^2 \tau}{i\omega + 1/\tau}$$

so that

$$\mathbf{D}(t) = \epsilon_0 \mathbf{E}(t) + \epsilon_0 \omega_p^2 \tau \left(\frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{i\omega} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega - \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{i\omega + 1/\tau} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega \right)$$

Both integrals enclosed between the large round brackets translate into convolutions following the convolution theorem:

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega = \int_{-\infty}^{\infty} f(t - t') \mathbf{E}(t') dt'$$

where $f(t)$ is the complementing component of the Fourier transform pair $(f(t), \hat{f}(\omega))$ [15].

Consider the first integral. The relevant Fourier transform pair in this case is:

$$\left(\theta(t), \frac{1}{i\omega} \right)$$

where $\theta(t)$ is the Heavyside step function, i.e.,

$$\theta(t) = \begin{cases} 0 & \text{for } t < 0 \\ 1 & \text{for } t \geq 0 \end{cases}$$

Consequently

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{i\omega} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega = \int_{-\infty}^{\infty} \theta(t - t') \mathbf{E}(t') dt' = \int_{-\infty}^t \mathbf{E}(t') dt'$$

because $\theta(t - t') = 0$ for $t' > t$ and 1 otherwise. Making use of the assumption that for $t < 0$ $\mathbf{E} = 0$ we get:

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{i\omega} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega = \int_0^t \mathbf{E}(t') dt'$$

Now let's figure out the second integral

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{i\omega + 1/\tau} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega$$

The relevant Fourier transform pair is

$$\left(\theta(t) e^{-t/\tau}, \frac{1}{i\omega + 1/\tau} \right)$$

Consequently

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{i\omega + 1/\tau} \hat{\mathbf{E}}(\omega) e^{i\omega t} d\omega = \int_{-\infty}^{\infty} \theta(t-t') e^{-(t-t')/\tau} \mathbf{E}(t') dt' = \int_0^t e^{-(t-t')/\tau} \mathbf{E}(t') dt'$$

where, as before, we made the assumption that $\mathbf{E} = 0$ for $t < 0$.

Adding all terms together we obtain

$$\mathbf{D}(t) = \epsilon_0 \mathbf{E}(t) + \epsilon_0 \omega_p^2 \tau \left(\int_0^t \mathbf{E}(t') dt' - \int_0^t e^{-(t-t')/\tau} \mathbf{E}(t') dt' \right)$$

Observe that both integrals yield correct units, because the units of $\omega_p^2 \tau dt$ cancel and so we get just $\epsilon_0 \mathbf{E}$ in both cases. Similarly $(t-t')/\tau$ is unitless.

Now the resulting Maxwell equations become:

$$\begin{aligned} \frac{\partial \mathbf{D}}{\partial t} &= \nabla \times \mathbf{H} \\ \mathbf{D}(t) &= \epsilon_0 \mathbf{E}(t) + \epsilon_0 \omega_p^2 \tau \left(\int_0^t \mathbf{E}(t') dt' - \int_0^t e^{-(t-t')/\tau} \mathbf{E}(t') dt' \right) \\ \frac{\partial \mathbf{H}}{\partial t} &= -\frac{1}{\mu_0} \nabla \times \mathbf{E} \end{aligned}$$

1.2 Elimination of ϵ_0 and μ_0

I'm going to show how to drop these pesky ϵ_0 and μ_0 . They are nasty two little numbers to have in computations:

$$\begin{aligned} \epsilon_0 &= 8.854185 \times 10^{-12} \text{ F/m} \\ \mu_0 &= 4\pi \times 10^{-7} \text{ H/m} \end{aligned}$$

The reason for their nastiness is that SI units are not "natural". We can get rid of both by doing the following:

$$\begin{aligned} \tilde{\mathbf{E}} &= \sqrt{\frac{\epsilon_0}{\mu_0}} \mathbf{E} \\ \tilde{\mathbf{D}} &= \sqrt{\frac{1}{\epsilon_0 \mu_0}} \mathbf{D} \\ c &= \sqrt{\frac{1}{\epsilon_0 \mu_0}} \text{ speed of light in vacuum} \end{aligned}$$

The resulting Maxwell equations are now:

$$\begin{aligned}\frac{\partial \tilde{\mathbf{D}}}{\partial t} &= c \nabla \times \mathbf{H} \\ \tilde{\mathbf{D}} &= \kappa \tilde{\mathbf{E}} \\ \frac{\partial \mathbf{H}}{\partial t} &= -c \nabla \times \tilde{\mathbf{E}}\end{aligned}$$

Replacing $t \rightarrow ct = \tilde{t}$ yields:

$$\begin{aligned}\frac{\partial \tilde{\mathbf{D}}}{\partial \tilde{t}} &= \nabla \times \mathbf{H} \\ \tilde{\mathbf{D}} &= \kappa \tilde{\mathbf{E}} \\ \frac{\partial \mathbf{H}}{\partial \tilde{t}} &= -\nabla \times \tilde{\mathbf{E}}\end{aligned}$$

where κ is the dimensionless parameter from $\epsilon = \epsilon_0 \kappa$. Because it is dimensionless, it doesn't change.

We are now going to drop the tilde from $\tilde{\mathbf{D}}$, $\tilde{\mathbf{E}}$ and \tilde{t} and we're going to rewrite our integral-differential system of equations for the Drude model:

$$\frac{\partial \mathbf{D}}{\partial t} = \nabla \times \mathbf{H} \quad (1)$$

$$\mathbf{D}(t) = \mathbf{E}(t) + \omega_p^2 \tau \left(\int_0^t \mathbf{E}(t') dt' - \int_0^t e^{-(t-t')/\tau} \mathbf{E}(t') dt' \right) \quad (2)$$

$$\frac{\partial \mathbf{H}}{\partial t} = -\nabla \times \mathbf{E} \quad (3)$$

where t , t' , dt' , ω_p and τ have to be recalculated in units of ct , i.e., in meters, or in some other more appropriate units of length. It's best to scale the unit of length to the problem at hand and make the grid spacing at the coarsest level be equal 1 unit of length. In these units light in vacuum propagates across one grid spacing in one unit of time.

1.3 Extracting \mathbf{E} from \mathbf{D}

In order to solve equations (1) through (3) we must be able to extract \mathbf{E} from equation (2). There are various ways to do this and here I am going to discuss three procedures.

The first two derive directly from

$$\mathbf{D}(t) = \mathbf{E}(t) + \omega_p^2 \tau \left(\int_0^t \mathbf{E}(t') dt' - \int_0^t e^{-(t-t')/\tau} \mathbf{E}(t') dt' \right)$$

which we can discretize as follows:

$$\mathbf{D}^n = \mathbf{E}^n + \omega_p^2 \tau \left(\sum_{k=0}^n \mathbf{E}^k \Delta t - \sum_{k=0}^n e^{-(n-k)\Delta t/\tau} \mathbf{E}^k \Delta t \right)$$

Observe that for $k = n$ the exponent function in the second integral becomes 1, so that the \mathbf{E}^n terms of both integrals cancel. Hence

$$\begin{aligned}\mathbf{D}^n &= \mathbf{E}^n + \omega_p^2 \tau \Delta t \left(\sum_{k=0}^{n-1} \mathbf{E}^k - \sum_{k=0}^{n-1} e^{-(n-k)\Delta t/\tau} \mathbf{E}^k \right) \\ &= \mathbf{E}^n + \omega_p^2 \tau \Delta t \left(\sum_{k=0}^{n-1} \mathbf{E}^k - e^{-\Delta t/\tau} \sum_{k=0}^{n-1} e^{-(n-1-k)\Delta t/\tau} \mathbf{E}^k \right)\end{aligned}$$

1.3.1 The Double Current Method

Define:

- $\omega_p^2 \tau \Delta t \sum_{k=0}^{n-1} \mathbf{E}^k = \mathbf{I}^{n-1}$
- $\omega_p^2 \tau \Delta t \sum_{k=0}^{n-1} e^{-(n-1-k)\Delta t/\tau} \mathbf{E}^k = \mathbf{S}^{n-1}$

Now observe that

$$\begin{aligned}\mathbf{E}^n &= \mathbf{D}^n - \mathbf{I}^{n-1} + e^{-\Delta t/\tau} \mathbf{S}^{n-1} \\ \mathbf{I}^n &= \mathbf{I}^{n-1} + \omega_p^2 \tau \Delta t \mathbf{E}^n \\ \mathbf{S}^n &= e^{-\Delta t/\tau} \mathbf{S}^{n-1} + \omega_p^2 \tau \Delta t \mathbf{E}^n\end{aligned}$$

These three equations provide us with the time-stepping procedure that lets us extract \mathbf{E} from \mathbf{D} at every time step and, at the same time, prepare \mathbf{I} and \mathbf{S} for the next time-step.

There is a substantial memory overhead in this computation. We have to maintain a copy of \mathbf{E} , \mathbf{D} , \mathbf{I} and \mathbf{S} , apart from \mathbf{H} , and this may become a burden on systems with limited memory.

1.3.2 The Single Current Method

It is possible to combine \mathbf{S} and \mathbf{I} into a single entity, let us call it \mathbf{M} , but in this case the computation becomes more complex and we have to use \mathbf{M}^{n-1} and \mathbf{M}^{n-2} in order to generate \mathbf{M}^n , so we don't gain anything in terms of memory savings and end up paying a higher computational price for it.

Nevertheless it is instructive to see how this comes about.

We start from

$$\mathbf{E}^n = \mathbf{D}^n - \mathbf{M}^{n-1}$$

where

$$\begin{aligned}\mathbf{M}^{n-1} &= \omega_p^2 \tau \Delta t \sum_{k=0}^n \left(1 - e^{-(n-k)\Delta t/\tau}\right) \mathbf{E}^k \\ &= \omega_p^2 \tau \Delta t \sum_{k=0}^{n-1} \left(1 - e^{-\Delta t/\tau} e^{-(n-1-k)\Delta t/\tau}\right) \mathbf{E}^k\end{aligned}$$

Similarly

$$\begin{aligned}\mathbf{M}^{n-2} &= \omega_p^2 \tau \Delta t \sum_{k=0}^{n-2} \left(1 - e^{-\Delta t/\tau} e^{-(n-2-k)\Delta t/\tau}\right) \mathbf{E}^k \\ \mathbf{M}^n &= \omega_p^2 \tau \Delta t \sum_{k=0}^n \left(1 - e^{-\Delta t/\tau} e^{-(n-k)\Delta t/\tau}\right) \mathbf{E}^k\end{aligned}$$

We are going to demonstrate that \mathbf{M}^n is a linear combination of \mathbf{E}^n , \mathbf{M}^{n-1} and \mathbf{M}^{n-2} :

$$\mathbf{M}^n = a\mathbf{E}^n + b\mathbf{M}^{n-1} + c\mathbf{M}^{n-2}$$

To simplify the notation we are going to use the following shortcuts:

- $\omega_p^2 \tau \Delta t = \gamma$
- $e^{-\Delta t/\tau} = x$

- $e^{-(n-2-k)\Delta t/\tau} = y_k$

Using these we can rewrite \mathbf{M}^n , \mathbf{M}^{n-1} and \mathbf{M}^{n-2} as follows:

$$\begin{aligned}\mathbf{M}^{n-2} &= \gamma \sum_{k=0}^{n-2} (1 - xy_k) \mathbf{E}^k \\ \mathbf{M}^{n-1} &= \gamma \left((1-x) \mathbf{E}^{n-1} + \sum_{k=0}^{n-2} (1 - x^2 y_k) \mathbf{E}^k \right) \\ \mathbf{M}^n &= \gamma \left((1-x) \mathbf{E}^n + (1-x^2) \mathbf{E}^{n-1} + \sum_{k=0}^{n-2} (1 - x^3 y_k) \mathbf{E}^k \right)\end{aligned}$$

Looking at \mathbf{M}^n we get that

$$a = \gamma(1-x)$$

because there is no \mathbf{E}^n in \mathbf{M}^{n-1} and \mathbf{M}^{n-2} .

Comparing the \mathbf{E}^{n-1} coefficients in \mathbf{M}^n and \mathbf{M}^{n-1} we find that

$$b(1-x) = 1 - x^2 = (1+x)(1-x) \quad \text{hence} \quad b = 1+x$$

Finally comparing the \mathbf{E}^k coefficients for $k < n-1$ we find that

$$b(1 - x^2 y_k) + c(1 - xy_k) = 1 - x^3 y_k$$

or, substituting $1+x$ in place of b :

$$\begin{aligned}(1+x)(1 - x^2 y_k) + c(1 - xy_k) \\ = 1 - x^2 y_k + x - x^3 y_k + c(1 - xy_k) \\ = 1 - x^3 y_k\end{aligned}$$

We see that we have to choose c so that

$$x - x^2 y_k + c(1 - xy_k) = 0$$

Selecting $c = -x$ does the trick.

In summary:

$$\mathbf{M}^n = \gamma(1-x) \mathbf{E}^n + (1+x) \mathbf{M}^{n-1} - x \mathbf{M}^{n-2}$$

or, restoring the original values of γ and x :

$$\mathbf{M}^n = \omega_p^2 \tau \Delta t \left(1 - e^{-\Delta t/\tau} \right) \mathbf{E}^k + \left(1 + e^{-\Delta t/\tau} \right) \mathbf{M}^{n-1} - e^{-\Delta t/\tau} \mathbf{M}^{n-2}$$

1.3.3 The Z-transform Method

The Z-transform method lets us work around the Fourier Transform formulas. It returns a result that is identical to the Single Current Method. But it is enlightening to see how this is done.

Our starting point is the formula:

$$\kappa(\omega) = 1 + \frac{\omega_p^2}{-\omega^2 + i\omega/\tau} = 1 + \frac{\omega_p^2 \tau}{i\omega} - \frac{\omega_p^2 \tau}{i\omega + 1/\tau}$$

Now we are going to take the Z -transform of κ , making use of the following Z -transform pairs table:

$$\begin{array}{ccc} 1 & \leftrightarrow & 1/\Delta t \\ \frac{1}{i\omega} & \leftrightarrow & \frac{1}{1-z^{-1}} \\ \frac{1}{i\omega + 1/\tau} & \leftrightarrow & \frac{1}{1-z^{-1}e^{-\Delta t/\tau}} \end{array}$$

The result is

$$\kappa(z) = \frac{1}{\Delta t} + \frac{\omega_p^2 \tau}{1-z^{-1}} - \frac{\omega_p^2 \tau}{1-z^{-1}e^{-\Delta t/\tau}}$$

The relationship between $\hat{\mathbf{D}}(\omega)$ and $\hat{\mathbf{E}}(\omega)$ is

$$\hat{\mathbf{D}}(\omega) = \kappa(\omega) \hat{\mathbf{E}}(\omega)$$

This translates into the convolution in the Z space:

$$\begin{aligned} \mathbf{D}(z) &= \kappa(z) \mathbf{E}(z) \Delta t \\ &= \mathbf{E}(z) + \omega_p^2 \tau \Delta t \left(\frac{1}{1-z^{-1}} - \frac{1}{1-z^{-1}e^{-\Delta t/\tau}} \right) \mathbf{E}(z) \\ &= \mathbf{E}(z) + \omega_p^2 \tau \Delta t \left(\frac{(1-e^{-\Delta t/\tau})z^{-1}}{1-(1+e^{-\Delta t/\tau})z^{-1}+e^{-\Delta t/\tau}z^{-2}} \right) \mathbf{E}(z) \\ &= \mathbf{E}(z) + z^{-1} \mathbf{M}(z) \end{aligned}$$

where

$$\mathbf{M}(z) = \omega_p^2 \tau \Delta t \left(\frac{(1-e^{-\Delta t/\tau})}{1-(1+e^{-\Delta t/\tau})z^{-1}+e^{-\Delta t/\tau}z^{-2}} \right) \mathbf{E}(z)$$

This yields

$$\mathbf{M}(z) \left(1 - (1+e^{-\Delta t/\tau})z^{-1} + e^{-\Delta t/\tau}z^{-2} \right) = \omega_p^2 \tau \Delta t \left(1 - e^{-\Delta t/\tau} \right) \mathbf{E}(z)$$

or

$$\mathbf{M}(z) = \left(1 + e^{-\Delta t/\tau} \right) z^{-1} \mathbf{M}(z) - e^{-\Delta t/\tau} z^{-2} \mathbf{M}(z) + \omega_p^2 \tau \Delta t \left(1 - e^{-\Delta t/\tau} \right) \mathbf{E}(z)$$

Switching from the Z -domain to the time domain yields:

$$\mathbf{M}^n = \left(1 + e^{-\Delta t/\tau} \right) \mathbf{M}^{n-1} - e^{-\Delta t/\tau} \mathbf{M}^{n-2} + \omega_p^2 \tau \Delta t \left(1 - e^{-\Delta t/\tau} \right) \mathbf{E}$$

where I have made use of the pairing between the Z and time domains:

$$\mathbf{M}^{n-k} \leftrightarrow z^{-k} \mathbf{M}(z)$$

The resulting formula is exactly the same as the one we obtained previously using the discretized convolution expression in the time domain.

1.3.4 The ADE Method

In this section I'm going to solve the equation:

$$\hat{\mathbf{D}}(\omega) = \hat{\mathbf{E}}(\omega) + \frac{\omega_p^2}{-\omega^2 + i\omega/\tau} \hat{\mathbf{E}}(\omega)$$

using the auxiliary differential equation method. As above, we introduce:

$$\hat{\mathbf{M}}(\omega) = \frac{\omega_p^2}{-\omega^2 + i\omega/\tau} \hat{\mathbf{E}}(\omega)$$

which yields

$$-\omega^2 \hat{\mathbf{M}}(\omega) + \frac{i\omega}{\tau} \hat{\mathbf{M}}(\omega) = \omega_p^2 \hat{\mathbf{E}}(\omega)$$

This time we convert this expression into an ordinary differential equation in the time-domain using the following table:

$$\begin{array}{ccc} -\omega^2 & \leftrightarrow & \frac{d^2}{dt^2} \\ i\omega & \leftrightarrow & \frac{d}{dt} \end{array}$$

In effect:

$$\frac{d^2 \mathbf{M}(t)}{dt^2} + \frac{1}{\tau} \frac{d\mathbf{M}(t)}{dt} = \omega_p^2 \hat{\mathbf{E}}(\omega)$$

We can discretize this equation as follows:

$$\begin{array}{ccc} \frac{d^2 \mathbf{M}(t)}{dt^2} & \leftrightarrow & \frac{\mathbf{M}^n - 2\mathbf{M}^{n-1} + \mathbf{M}^{n-2}}{\Delta t^2} \\ \frac{d\mathbf{M}(t)}{dt} & \leftrightarrow & \frac{\mathbf{M}^n - \mathbf{M}^{n-2}}{2\Delta t} \end{array}$$

So that after the substitution we obtain

$$\frac{\mathbf{M}^n - 2\mathbf{M}^{n-1} + \mathbf{M}^{n-2}}{\Delta t^2} + \frac{1}{\tau} \frac{\mathbf{M}^n - \mathbf{M}^{n-2}}{2\Delta t} = \omega_p^2 \mathbf{E}^{n-1}$$

or:

$$\mathbf{M}^n \left(\frac{\tau}{\Delta t} + \frac{1}{2} \right) = \frac{2\tau}{\Delta t} \mathbf{M}^{n-1} + \left(\frac{\tau}{\Delta t} - \frac{1}{2} \right) \mathbf{M}^{n-2} + \omega_p^2 \tau \Delta t \mathbf{E}^{n-1}$$

Dividing both sides by $\frac{\tau}{\Delta t} + \frac{1}{2}$ yields after some simplifications and rearrangements:

$$\mathbf{M}^n = \frac{2}{1 + \frac{\Delta t}{2\tau}} \mathbf{M}^{n-1} - \frac{1 - \frac{\Delta t}{2\tau}}{1 + \frac{\Delta t}{2\tau}} \mathbf{M}^{n-2} + \frac{\omega_p^2 \tau \Delta t}{\frac{\tau}{\Delta t} + \frac{1}{2}} \mathbf{E}^{n-1}$$

Now let us have a closer look at each term on the right hand side in the limit $\delta t/\tau \rightarrow 0$. First the \mathbf{M}^{n-1} term:

$$\begin{aligned} & \frac{2}{1 + \frac{\Delta t}{2\tau}} \mathbf{M}^{n-1} \\ & \approx 2 \left(1 - \frac{\Delta t}{2\tau} \right) \mathbf{M}^{n-1} \\ & = \left(1 + 1 - \frac{\Delta t}{\tau} \right) \mathbf{M}^{n-1} \\ & \approx \left(1 + e^{-\Delta t/\tau} \right) \mathbf{M}^{n-1} \end{aligned}$$

Now the \mathbf{M}^{n-2} term:

$$\begin{aligned}
& -\frac{1 - \frac{\Delta t}{2\tau}}{1 + \frac{\Delta t}{2\tau}} \mathbf{M}^{n-2} \\
& \approx -\left(1 - \frac{\Delta t}{2\tau}\right) \left(1 - \frac{\Delta t}{2\tau}\right) \mathbf{M}^{n-2} \\
& = -\left(1 - \frac{\Delta t}{\tau} + \frac{\Delta t^2}{4\tau^2}\right) \mathbf{M}^{n-2} \\
& \approx -\left(1 - \frac{\Delta t}{\tau}\right) \mathbf{M}^{n-2} \\
& \approx -e^{-\Delta t/\tau} \mathbf{M}^{n-2}
\end{aligned}$$

And finally the \mathbf{E}^{n-1} term:

$$\begin{aligned}
& \frac{\omega_p^2 \tau \Delta t}{\frac{\tau}{\Delta t} + \frac{1}{2}} \mathbf{E}^{n-1} \\
& \approx \omega_p^2 \tau \Delta t \frac{\Delta t}{\tau} \frac{1}{1 + \frac{\Delta t}{2\tau}} \mathbf{E}^{n-1} \\
& \approx \omega_p^2 \tau \Delta t \frac{\Delta t}{\tau} \left(1 - \frac{\Delta t}{2\tau}\right) \mathbf{E}^{n-1} \\
& = \omega_p^2 \tau \Delta t \left(\frac{\Delta t}{\tau} - \frac{\Delta t^2}{2\tau^2}\right) \mathbf{E}^{n-1} \\
& \approx \omega_p^2 \tau \Delta t \frac{\Delta t}{\tau} \mathbf{E}^{n-1} \\
& = \omega_p^2 \tau \Delta t \left(1 - \left(1 - \frac{\Delta t}{\tau}\right)\right) \mathbf{E}^{n-1} \\
& \approx \omega_p^2 \tau \Delta t \left(1 - e^{-\Delta t/\tau}\right) \mathbf{E}^{n-1}
\end{aligned}$$

Now we combine all this together to get:

$$\mathbf{M}^n \approx \left(1 + e^{-\Delta t/\tau}\right) \mathbf{M}^{n-1} - e^{-\Delta t/\tau} \mathbf{M}^{n-2} + \omega_p^2 \tau \Delta t \left(1 - e^{-\Delta t/\tau}\right) \mathbf{E}^{n-1}$$

and this is again the same single current model expression we've seen before.

But it is important to remember that we get the agreement in the $\Delta t/\tau \rightarrow 0$ limit only. This provides us with an additional condition on Δt .

1.3.5 Code Implementation

All three formulas, i.e., the double current, the single current and the ADE one, are implemented in the code within subroutine `d_to_e`:

```

subroutine d_to_e (
&   imin, imax, jmin, jmax,
&   dx, dy, ex, ey, ix, iy, sx, sy,
&   x0, delta_x, y0, delta_y, t, delta_t,
&   omega_p, tau, model, xa, xb, ya, yb,
&   current_model, verbose)
implicit none

```

```

integer imin, imax, jmin, jmax
double precision
&    dx(imin:imax, jmin:jmax),
&    dy(imin:imax, jmin:jmax),
&    ex(imin:imax, jmin:jmax),
&    ey(imin:imax, jmin:jmax),
&    ix(imin:imax, jmin:jmax),
&    iy(imin:imax, jmin:jmax),
&    sx(imin:imax, jmin:jmax),
&    sy(imin:imax, jmin:jmax)
double precision
&    x0, delta_x, y0, delta_y, t, delta_t, omega_p, tau,
&    xa, xb, ya, yb
integer model, current_model, verbose

```

where fields `ix`, `iy`, `sx` and `sy` play a double role, i.e., either of I_x, I_y, S_x, S_y as used in the double current model, or of S^n, S^{n-1} , of single current models.

A specific current model is selected by setting `current_model` to 1 (double current), 2 (single current, ADE) or 3 (single current, exact formula).

If the selected current model is 1 then the computation looks as follows for D_x :

```

coeff_1 = exp(-delta_t/tau)
coeff_2 = omega_p**2 * tau * delta_t
...
do j = jmin, jmax
  y = y0 + j * delta_y - delta_y_by_2
  do i = imin, imax
    x = x0 + i * delta_x
    if (current_model .eq. 1) then
      coeff_0 = distrib(x, y, local_model, xa, xb, ya, yb)
      delta_s = coeff_1 * sx(i, j)
      ex(i, j) = dx(i, j) - ix(i, j) + delta_s
      delta_i = coeff_0 * coeff_2 * ex(i, j)
      ix(i, j) = ix(i, j) + delta_i
      sx(i, j) = delta_s + delta_i
    else ...
  end do
end do

```

where `coeff_0` is a distribution function that is equal to 1 where metal is present, to 0 where it is absent and to anything in between on the metal-vacuum border, according to the border model. Thus, within the metal medium the computation evaluates:

$$\begin{aligned}
\Delta S &= e^{-\Delta t/\tau} S_x^{n-1} \\
E_x^n &= D_x^n - I_x^{n-1} + \Delta S = D_x^n - I_x^{n-1} + e^{-\Delta t/\tau} S_x^{n-1} \\
\Delta I &= \omega_p^2 \tau \Delta t E_x^n \\
I_x^n &= I_x^{n-1} + \Delta I = I_x^{n-1} + \omega_p^2 \tau \Delta t E_x^n \\
S_x^n &= \Delta S + \Delta I = e^{-\Delta t/\tau} S_x^{n-1} + \omega_p^2 \tau \Delta t E_x^n
\end{aligned}$$

in accordance with formulas derived in section 1.3.1.

The computation is similar for D_y with the only difference being the location of the D_y field:

```

do j = jmin, jmax
  y = y0 + j * delta_y
  do i = imin, imax
    x = x0 + i * delta_x - delta_x_by_2

```

For current model 2 (single current, ADE) the computation looks as follows:

```

      else if (current_model .eq. 2) then
        ex(i, j) = dx(i, j) - ix(i, j)
        hold = 2.0/(1.0 + delta_t_by_2_tau) * ix(i, j)
&        - (1.0 - delta_t_by_2_tau)/(1.0 + delta_t_by_2_tau)
&        * sx(i, j)
&        + distrib(x, y, local_model, xa, xb, ya, yb)
&        * coeff_2 / (tau_by_delta_t + 0.5) * ex(i, j)
        sx(i, j) = ix(i, j)
        ix(i, j) = hold
      else ...

```

which, remembering that now ix stands for S_x^{n-1} and sx stands for S_x^{n-2} , translates into:

$$\begin{aligned}
E_x^n &= D_x^n - S_x^{n-1} \\
\text{hold} &= \frac{2}{1 + \frac{\Delta t}{2\tau}} S_x^{n-1} - \frac{1 - \frac{\Delta t}{2\tau}}{1 + \frac{\Delta t}{2\tau}} S_x^{n-2} + \frac{\omega_p^2 \tau \Delta t}{\frac{\tau}{\Delta t} + 1/2} E_x^n \\
S_x^{n-2} &\leftarrow S_x^{n-1} \\
S_x^{n-1} &\leftarrow \text{hold}
\end{aligned}$$

in accordance with formulas derived in section 1.3.4.

Finally, for current model 3, which is a single-current exact formula model, the computation looks as follows:

```

      else if (current_model .eq. 3) then
        ex(i, j) = dx(i, j) - ix(i, j)
        hold = (1.0 + coeff_1) * ix(i, j)
&        - coeff_1 * sx(i, j)
&        + distrib(x, y, local_model, xa, xb, ya, yb)
&        * coeff_2 * (1.0 - coeff_1) * ex(i, j)
        sx(i, j) = ix(i, j)
        ix(i, j) = hold
      end if

```

which translates to:

$$\begin{aligned}
E_x^n &= D_x^n - S_x^{n-1} \\
\text{hold} &= \left(1 + e^{-\Delta t/\tau}\right) S_x^{n-1} - e^{-\Delta t/\tau} S_x^{n-2} + \omega_p^2 \tau \Delta t \left(1 - e^{-\Delta t/\tau}\right) E_x^n \\
S_x^{n-2} &\leftarrow S_x^{n-1} \\
S_x^{n-1} &\leftarrow \text{hold}
\end{aligned}$$

which matches formulas derived in section 1.3.2.

Observe that where `distrib` returns zero, no accumulation takes place and ix , iy , sx and sy all remain zero and thus do not contribute to \mathbf{E} . In this case we simply get that $\mathbf{E} = \mathbf{D}$, since in our units $\epsilon_0 = 1$.

1.3.6 Summary

I have derived three formulas for extracting \mathbf{E}^n from \mathbf{D}^n : the double current one, the single current formula, which was derived twice – first using straightforward algebra and then using the Z transform, and finally the single current formula, which was derived using the auxiliary differential equation method. The latter agrees with the Z transform formula in the limit $\Delta t/\tau \rightarrow 0$. All three formulas have the same memory cost, i.e., one has to store two additional auxiliary fields apart from \mathbf{D} and \mathbf{E} , but the double current formula is the cheapest computationally, while it should be as accurate as the Z -transform derived single current formula. The ADE single current formula is computationally cheaper than the Z -transform formula, because it does not rely on exponentials, but it is less accurate. The Z -transform or the double current formulas are accurate, with the only error creeping in through the discretization of the convolution integrals in the time-domain.

A look at the Z -transform single current formula or at the double current formula shows us that none should lead to destabilization of the computational procedure, because the exponents that appears in both formulas is always damping. Physically this means that any signal that enters the medium, a metal in this case, must be quickly attenuated. This will result in a reflection off the medium and, if the medium is so shaped, possible focusing of the signal, which may result in a local signal enhancement. But this is all there is to it.

If the frequency of the signal exceeds plasma frequency of the metal, the metal becomes transparent to the signal. This, of course, is a good test for the program – clearly we should be able to observe the same effect in simulations and... we *do*!

Experimental tests of all three formulas showed that there was no visible difference between them, i.e., in practical runs with appropriately chosen Δt they returned identical results for all situations tested and for all metal-border distribution models.

1.4 Medium Parameters in Natural Units

How to calculate medium parameters in natural units for use with subroutine `d_to_e`?

Medium parameters for metals are given in terms of two frequencies: the plasma frequency, ω_p , and the attenuation frequency, $1/\tau$. Because in all calculations we take $c = 1$, time and space units are the same. In turn, space units are determined by

1. $\Delta x = \Delta y = 1^1$
2. Physical wavelength of a harmonic wave or width of the incident pulse, and
3. resolution required to resolve the wave or a pulse adequately.

In order to convert ω_p and τ to *natural units* we simply have to find what these units are for a given system defined by (1) through (3) above.

We begin by defining (3). This is best done by experimenting with waves of various length in context of wave injection into and its subsequent extraction from the total field region, as discussed in sections 3 and 3.1. It is stated there that the incident signal is calculated analytically on the total field region boundary, but propagated numerically within the region. This results in a slight mismatch, which produces a tiny wake within the scattered field region that accompanies the numerically propagated wave. The better the resolution of the incident wave and the more accurate the computation, the smaller is the wake in the scattered field region.

Although one can find statements in various learned books that one can resolve waves with just a few points, these statements derive from the CFL stability criterion. But for us here it is not enough

¹Although the programs developed within this project allow for varying Δx and Δy independently, this is neither advised nor has it been tested.

that we are in a stable regime. We want to be accurate too. And so, through multiple experiments I arrived at the resolution of 40, i.e., when the wavelength is 40, the parasitic wake in the scattered field region is hardly visible.

The way we are going to figure out material parameters is going to derive from this assumption, that a wavelength of a wave is 40 units of length. This is fixed. What is not fixed is the wavelength itself.

Suppose we consider a neon laser light of $\lambda = 540.06$ nm, then the unit of length is:

$$\Delta x = \Delta y = 540.06 \text{ nm} / 40 \approx 13.5 \text{ nm}$$

The unit² of time, Δt , is then time in which a wave front passes the distance of 13.5 nm, i.e.,

$$\Delta t = \frac{\Delta x}{c} = \frac{13.5 \times 10^{-9} \text{ m}}{2.997925 \times 10^8 \text{ m/s}} \approx 4.5 \times 10^{-17} \text{ s}$$

Now, consider silver, for which we have:

$$\begin{aligned} 1/\tau &= 57 \text{ THz} \\ f_p &= 1/T_p = 2000 \text{ THz} \end{aligned}$$

Let us begin with τ :

$$\begin{aligned} \tau &= \frac{1}{57 \times 10^{12} \text{ 1/s}} = .0175438596 \times 10^{12} \text{ s} = .0175438596 \times 10^{12} \frac{\text{s}}{\Delta t} \Delta t \\ &= \frac{.0175438596 \times 10^{12} \text{ s}}{4.5 \times 10^{-17} \text{ s}} \Delta t \approx 389.9 \Delta t \end{aligned}$$

In turn, we get the following for ω_p :

$$\begin{aligned} \omega_p &= \frac{2\pi}{T_p} = \frac{2\pi}{T_p/\Delta t} \frac{1}{\Delta t} = \frac{2\pi\Delta t}{T_p} \frac{1}{\Delta t} \\ &= \frac{2 \times 3.14 \times 4.5 \times 10^{-17}}{T_p} \frac{1}{\Delta t} = f_p \times 2 \times 3.14 \times 4.5 \times 10^{-17} \frac{1}{\Delta t} \\ &= 2,000 \times 10^{12} \times 2 \times 3.14 \times 4.5 \times 10^{-17} \frac{1}{\Delta t} \approx 0.5652 \frac{1}{\Delta t} \end{aligned}$$

In summary:

$$\begin{aligned} \lambda &= 540.06 \text{ nm} \\ \Delta x &= 13.5 \text{ nm} \\ \Delta t &= 4.5 \times 10^{-17} \text{ s} \\ \omega_p &= 0.5652 \frac{1}{\Delta t} \\ \tau &= 389.9 \Delta t \end{aligned}$$

In this case the incident wave is going to reflect from the silver boundary.

²Nota bene: in this section Δt is the unit of time, not the length of the time step. Similarly Δx is the unit of length, not the grid constant.

In a similar way one can show that for the same material parameters, i.e., silver, if the incident light is deep ultraviolet, e.g., $\lambda = 40$ nm, then:

$$\begin{aligned}\lambda &= 40 \text{ nm} \\ \Delta x &= 1 \text{ nm} \\ \Delta t &= 3.336 \times 10^{-18} \text{ s} \\ \omega_p &= 0.042 \frac{1}{\Delta t} \\ \tau &= 5257.794 \Delta t\end{aligned}$$

In this case the incident wave is going to pass through the silver medium with very little absorption.

The above illustrates the general strategy for computations adopted within this project. The wave is resolved in some optimal way, e.g., it is spread over 40 level 0 grid points, and this is fixed, regardless of the wavelength. For a given material we are now going to recalculate its material parameters in natural units and use the results in computations. If we change the wavelength of the incident light, the new system still resolves the wave on 40 level 0 grid points, but the material parameters have to change even though we continue to work with the same material.

Because changing the wavelength of incident light changes the unit of length, we may have to enlarge the grid and adjust the size of metal structures accordingly too. Their physical size is not going to change, but their size expressed in the new units of length will be different.

2 PML ABCs

The TE equations in the frequency domain look as follows:

$$\begin{aligned}i\omega D_x &= \partial_y H_z \\ i\omega D_y &= -\partial_x H_z \\ D_x &= \epsilon E_x \\ D_y &= \epsilon E_y \\ i\omega H_z &= \partial_y E_x - \partial_x E_y\end{aligned}$$

In order to absorb incident signal within a thin boundary layer we introduce three functions of position, β_x , β_y and α_z , inserting them in the frequency domain equations as follows:

$$\begin{aligned}i\omega D_x \beta_x(x) \beta_y(y) &= \partial_y H_z \\ i\omega D_y \beta_y(x) \beta_x(y) &= -\partial_x H_z \\ D_x &= \epsilon E_x \\ D_y &= \epsilon E_y \\ i\omega H_z \alpha_z(x) \alpha_z(y) &= \partial_y E_x - \partial_x E_y\end{aligned}$$

Functions β_x , β_y and α_z form PML ABCs in TE if [11] [13]

$$\begin{aligned}\beta_x &= 1/\beta_y \\ \alpha_z &= \beta_y\end{aligned}$$

Without much loss in generality we can assume the following form for $\alpha_z = \beta_y = 1/\beta_x$ [12] [13]:

$$\alpha_z = 1 + \frac{\sigma}{i\omega}$$

$$\begin{aligned}\beta_x &= \frac{1}{1 + \frac{\sigma}{i\omega}} \\ \beta_y &= 1 + \frac{\sigma}{i\omega}\end{aligned}$$

where σ is a function of depth *into* the PML layer from within the computational domain. It is equal 0 within the computational domain.

With these in place Maxwell equations in the frequency domain look as follows:

$$\begin{aligned}i\omega D_x \left(1 + \frac{\sigma(y)}{i\omega}\right) &= \left(1 + \frac{\sigma(x)}{i\omega}\right) \partial_y H_z \\ i\omega D_y \left(1 + \frac{\sigma(x)}{i\omega}\right) &= -\left(1 + \frac{\sigma(y)}{i\omega}\right) \partial_x H_z \\ D_x &= \epsilon E_x \\ D_y &= \epsilon E_y \\ i\omega H_z \left(1 + \frac{\sigma(x)}{i\omega}\right) \left(1 + \frac{\sigma(y)}{i\omega}\right) &= \partial_y E_x - \partial_x E_y\end{aligned}$$

Converting from the frequency to the time domain yields

$$\begin{aligned}\partial_t D_x + \sigma(y) D_x &= \partial_y H_z + \sigma(x) \int_0^t \partial_y H_z dt' \\ \partial_t D_y + \sigma(x) D_y &= -\partial_x H_z - \sigma(y) \int_0^t \partial_x H_z dt' \\ D_x &= \epsilon E_x \\ D_y &= \epsilon E_y \\ \partial_t H_z + \sigma(x) H_z + \sigma(y) H_z &= \partial_y E_x - \partial_x E_y - \sigma(x)\sigma(y) \int_0^t H_z dt'\end{aligned}$$

The product of two sigmas, $\sigma(x)\sigma(y)$, vanishes everywhere with the exception of the corners, where *both* $\sigma(x)$ and $\sigma(y)$ are different from zero. Furthermore, for an oscillating H_z the integral $\int_0^t H_z dt'$ is going to be zero on average. Consequently, I am going to neglect this term altogether in these computations, so that the last equation simplifies to:

$$\partial_t H_z + \sigma(x) H_z + \sigma(y) H_z = \partial_y E_x - \partial_x E_y$$

In leap-frog discretization of these time-domain equations, we evaluate non-differentiated terms on the left hand side at the same time slice as the derivatives and as the right hand side. This results in the replacement of, e.g., D_x^n with $(D_x^n + D_x^{n-1})/2$ – with the following effect:

$$\begin{aligned}D_x^n &= D_x^{n-1} \frac{1 - \sigma(y)\Delta t/2}{1 + \sigma(y)\Delta t/2} + \frac{\Delta t}{1 + \sigma(y)\Delta t/2} \left(\partial_y H_z + \frac{\sigma(x)\Delta t}{2} \sum 2\partial_y H_z \right) \\ D_y^n &= D_y^{n-1} \frac{1 - \sigma(x)\Delta t/2}{1 + \sigma(x)\Delta t/2} - \frac{\Delta t}{1 + \sigma(x)\Delta t/2} \left(\partial_x H_z + \frac{\sigma(y)\Delta t}{2} \sum 2\partial_x H_z \right) \\ H_z^n &= H_z^{n-1} \frac{1 - \sigma(x)\Delta t/2 - \sigma(y)\Delta t/2}{1 + \sigma(x)\Delta t/2 + \sigma(y)\Delta t/2} + \frac{\Delta t}{1 + \sigma(x)\Delta t/2 + \sigma(y)\Delta t/2} (\partial_y E_x - \partial_x E_y)\end{aligned}$$

The last equation, for H_z^n can be rewritten as

$$H_z^n = H_z^{n-1} \left(\frac{1 - \sigma(x)\Delta t/2}{1 + \sigma(x)\Delta t/2} \right) \left(\frac{1 - \sigma(y)\Delta t/2}{1 + \sigma(y)\Delta t/2} \right) + \Delta t \left(\frac{1}{1 + \sigma(x)\Delta t/2} \right) \left(\frac{1}{1 + \sigma(y)\Delta t/2} \right) (\partial_y E_x - \partial_x E_y)$$

within the $\mathcal{O}(\Delta t)^2$ accuracy.

Following [13] I replace $\sigma(x)\Delta t/2$ with

$$\frac{\sigma(x)\Delta t}{2} = f(x) = \frac{1}{3} \left(\frac{\text{depth into the PML layer}}{\text{width of the PML layer}} \right)^3$$

This implies that when carrying out computations within the PML layer we have to evaluate one of the following three cases:

1. $f(x)$
2. $1/(1 + f(x))$
3. $(1 - f(x))/(1 + f(x))$

The following Fortran function defined on `update.f` implements this computation:

```
function pml(x, xmin, x1, x2, xmax, mode)
implicit none
double precision pml
double precision x, xmin, x1, x2, xmax
integer mode
```

where x is the current position, x_{\min} is the left edge of the computational domain, x_{\max} is the right edge of the computational domain, x_1 is where the PML layer begins on the left hand side ($x_{\min} < x_1$), x_2 is where the PML layer begins on the right hand side ($x_2 < x_{\max}$ and $x_1 < x_2$) and `mode` is an integer, 1, 2, or 3, that specifies whether we want to evaluate $f(x)$ (1), $1/(1 + f(x))$ (2) or $(1 - f(x))/(1 + f(x))$ (3). If $x_1 < x < x_2$ the function returns 0 for mode 1 or 1 for modes 2 and 3 right away, which makes it relatively inexpensive to call outside of the PML region.

Of course, this function can be also called with y in place of x .

The computation then unfolds as follows. For H_z we get:

```
dt_by_dy = delta_t / delta_y
dt_by_dx = delta_t / delta_x

do j = jmin + 1, jmax - 1
  y = y0 + j * delta_y
  do i = imin + 1, imax - 1
    x = x0 + i * delta_x
    hz(i, j) =
&      pml(x, xmin, x1, x2, xmax, 3)
&      * pml(y, ymin, y1, y2, ymax, 3)
&      * hz(i, j)
&      + pml(x, xmin, x1, x2, xmax, 2)
&      * pml(y, ymin, y1, y2, ymax, 2)
&      * ( - (ey(i + 1, j) - ey(i, j)) * dt_by_dx
&      + (ex(i, j + 1) - ex(i, j)) * dt_by_dy)
  end do
end do
```

For D_x :

```
do j = jmin + 1, jmax - 1
  y = y0 + j * delta_y - delta_y / 2
  do i = imin + 1, imax - 1
    x = x0 + i * delta_x
```

```

        delta_hz = hz(i, j) - hz(i, j - 1)
        idx(i, j) = idx(i, j)
&          + 2 * pml(x, xmin, x1, x2, xmax, 1) * delta_hz
        dx(i, j) =
&          pml(y, ymin, y1, y2, ymax, 3) * dx(i, j)
&          + pml(y, ymin, y1, y2, ymax, 2)
&          * (delta_hz + idx(i, j)) * dt_by_dy
    end do
end do

```

where idx is $\frac{\sigma(x)\Delta t}{2} \sum 2\partial_y H_z \Delta y$. When $(\text{delta_hz} + \text{idx}(i, j))$ is multiplied by $\Delta t/\Delta y$ in the last step, the correct formula is obtained both for $\nabla \times \mathbf{H}$ and for the accumulation term.

Finally, for D_y :

```

do j = jmin + 1, jmax - 1
  y = y0 + j * delta_y
  do i = imin + 1, imax - 1
    x = x0 + i * delta_x - delta_x / 2
    delta_hz = hz(i, j) - hz(i - 1, j)
    idy(i, j) = idy(i, j)
&          + 2 * pml(y, ymin, y1, y2, ymax, 1) * delta_hz
    dy(i, j) =
&          pml(x, xmin, x1, x2, xmax, 3) * dy(i, j)
&          - pml(x, xmin, x1, x2, xmax, 2)
&          * (delta_hz + idy(i, j)) * dt_by_dx
  end do
end do

```

where idy is $\frac{\sigma(y)\Delta t}{2} \sum 2\partial_x H_z \Delta x$. When $(\text{delta_hz} + \text{idy}(i, j))$ is multiplied by $\Delta t/\Delta x$ in the last step, the correct formul is obtained both for $\nabla \times \mathbf{H}$ and for the accumulation term.

Observe that x and y correspond to the positions in the grid at which a field is defined. In this code H_z is cell-centered, whereas D_x and D_y are staggered.

The above implementations may not be the most efficient in terms of speed of execution, but at this stage I believe code clarity and code correctness are more important³.

This computation, implemented by subroutines `update_d_pml` and `update_h_pml`, is invoked *only* within level 0, because only level 0 fields reach beyond the total field region and towards the PML boundaries. Fields within higher levels call simpler subroutines `update_h` and `update_d`, which implement plain Maxwell equations *without* PML ABC terms.

3 Injection and Extraction of the Incident Field

The incident field is injected into the total field region and then extracted from it by the means of tweaking appropriate field derivatives on the border of the region. An analytical formula is used to propagate the field along the boundary, whereas propagation of the field within the total field region is numerical. This has the advantage of letting us observe the effects of numerical dispersion and test the accuracy of the solution method – covering also the accuracy of multi-level computations.

On the other hand this has a disadvantage as well, because due to numerical dispersion analytically

³I do not claim the code to be correct, but clear coding makes it easier to implement correct computation in the first place and then to spot possible errors if any have been made.

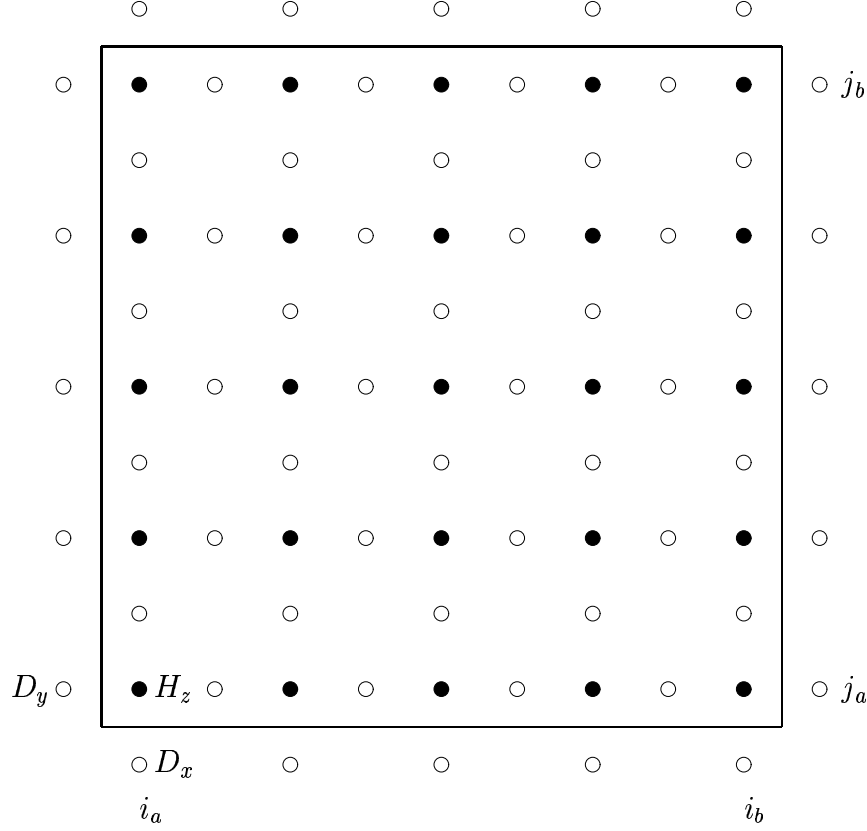


Figure 1: Definition of total and scattered field regions laid out on top of the staggered grid. The total field region is within the rectangle defined by (i_a, j_a) and (i_b, j_b) . Everything outside this rectangle is the scattered field region. Magnetic field H_z is cell centered (black dots). Field D_y is side-centered to the left of H_z and field D_x is side-centered below H_z , i.e., $H_z(i, j) \equiv H_z(x_0 + i\Delta x, y_0 + j\Delta y)$, $D_x(i, j) \equiv D_x(x_0 + i\Delta x, y_0 + j\Delta y - \Delta y/2)$ and $D_y(i, j) \equiv D_y(x_0 + i\Delta x - \Delta x/2, y_0 + j\Delta y)$.

and numerically propagated pulses diverge eventually, which results in a small unsubtracted signal within the scattered field region, especially towards the end of pulse or wave propagation within the total field region. But this signal is absorbed by PML ABCs, and does not enter the total field region. It can be minimized by resolving the wave better and by injecting the wave into the total field region gradually.

Figure 1 illustrates the total/scattered field regions, with the total field region enclosed within a rectangle defined by (i_a, j_a) and (i_b, j_b) .

Maxwell equations

$$\begin{aligned}\partial_t \mathbf{D} &= \nabla \times \mathbf{H} \\ \partial_t \mathbf{H} &= -\nabla \times \mathbf{E}\end{aligned}$$

discretize in the TE mode as follows:

$$\begin{aligned}D_x^n(i, j) &= D_x^{n-1}(i, j) + \frac{H_z^{n-1}(i, j) - H_z^{n-1}(i, j-1)}{\Delta y} \Delta t \\ D_y^n(i, j) &= D_y^{n-1}(i, j) - \frac{H_z^{n-1}(i, j) - H_z^{n-1}(i-1, j)}{\Delta x} \Delta t\end{aligned}$$

$$H_z^n(i, j) = H_z^{n-1}(i, j) - \left(\frac{E_y^n(i+1, j) - E_y^n(i, j)}{\Delta x} - \frac{E_x^n(i, j+1) - E_x^n(i, j)}{\Delta y} \right) \Delta t$$

where $\mathbf{D}^n \equiv \mathbf{D}(t_0 + n\Delta t)$ and $\mathbf{H}^n \equiv \mathbf{H}(t_0 + n\Delta t + \Delta t/2)$.

Now consider update of $D_y(i_a, j)$ for $j_a \leq j \leq j_b$:

$$D_y^n(i_a, j) = D_x^{n-1}(i_a, j) - \frac{H_z^{n-1}(i_a, j) - H_z^{n-1}(i_a - 1, j)}{\Delta x} \Delta t$$

Here $H_z^{n-1}(i_a, j)$ is *inside* the total field region but $D_y^n(i_a, j)$ and $H_z^{n-1}(i_a - 1, j)$ are *outside*. The inside value of $H_z^{n-1}(i_a, j)$ has the incident field component, which we must subtract from it in order for the update to make sense. But this subtraction can be carried out afterwards by a separate subroutine, which will have to carry out:

$$D_y^n(i_a, j) \leftarrow D_y^n(i_a, j) + H_z^{\text{incident}}(i_a, j, n-1) \Delta t / \Delta x$$

for all $j \in [j_a, j_b]$.

Similarly for $D_y(i_b + 1, j)$ for all $j_a \leq j \leq j_b$:

$$D_y^n(i_b + 1, j) = D_x^{n-1}(i_b + 1, j) - \frac{H_z^{n-1}(i_b + 1, j) - H_z^{n-1}(i_b, j)}{\Delta x} \Delta t$$

Here $D_y^n(i_b + 1, j)$ and $H_z^{n-1}(i_b + 1, j)$ are *outside*, but $H_z^{n-1}(i_b, j)$ is *inside* the total field region and so it carries the incident field component, which we must subtract from it in order for the update to be meaningful. This can be done again by a separate subroutine, which will have to carry out:

$$D_y^n(i_b + 1, j) \leftarrow D_y^n(i_b + 1, j) - H_z^{\text{incident}}(i_b, j, n-1) \Delta t / \Delta x$$

The Fortran subroutine that makes these corrections is called `inject_d`. It is a somewhat convoluted subroutine, but what makes it so is not the computation itself, but figuring out whether it needs to be done for a given range of i and j values at all, since the subroutine is going to be invoked on a plaquette, i.e., on some subset of the computational domain, in a multi-processor version of the code and this includes the Chombo formulation, even if run on a single processor.

In any case, this is what the relevant portion of the code looks like:

```

      if ((imin .lt. ia) .and. (ia .lt. imax)) then
        do j = jfrom, jto
          y = y0 + j * delta_y
          hz_incident = pulse(y, t - delta_t * 0.5,
&              t0, sigma, mode)
          dy(ia, j) = dy(ia, j) + hz_incident * dt_by_dx
        end do
      end if
      if ((imin .lt. ib + 1) .and. (ib + 1 .lt. imax)) then
        do j = jfrom, jto
          y = y0 + j * delta_y
          hz_incident = pulse(y, t - delta_t * 0.5,
&              t0, sigma, mode)
          dy(ib + 1, j) = dy(ib + 1, j) - hz_incident * dt_by_dx
        end do
      end if

```

Now, let us consider, in a similar way, corrections needed to update $D_x(i, j_a)$ and $D_x(i, j_b + 1)$ for all $i \in [i_a, i_b]$. We begin with $D_x(i, j_a)$. Here we have:

$$D_x^n(i, j_a) = D_x^{n-1}(i, j_a) + \frac{H_z^{n-1}(i, j_a) - H_z^{n-1}(i, j_a - 1)}{\Delta y} \Delta t$$

where $D_x^n(i, j_a)$ and $H_z^{n-1}(i, j_a - 1)$ are *outside* the total field region, but $H_z^{n-1}(i, j_a)$ is *inside* it. Consequently, we will have to subtract the incident field contribution from $H_z^{n-1}(i, j_a)$ in evaluating $D_x^n(i, j_a)$ for $i \in [i_a, i_b]$. The resulting formula is:

$$D_x^n(i, j_a) \leftarrow D_x^n(i, j_a) - H_z^{\text{incident}}(i, j_a, n - 1) \Delta t / \Delta y$$

and for $j = j_b + 1$:

$$D_x^n(i, j_b + 1) = D_x^{n-1}(i, j_b + 1) + \frac{H_z^{n-1}(i, j_b + 1) - H_z^{n-1}(i, j_b)}{\Delta y} \Delta t$$

where this time it is $H_z^{n-1}(i, j_b)$ that is inside the total field region. So the corrective formula is going to be:

$$D_x^n(i, j_b + 1) \leftarrow D_x^n(i, j_b + 1) + H_z^{\text{incident}}(i, j_b, n - 1) \Delta t / \Delta y$$

Here is how these two corrections are coded within `inject_d`:

```

if ((jmin .lt. ja) .and. (ja .lt. jmax)) then
  ya = y0 + ja * delta_y
  hz_incident = pulse(ya, t - delta_t * 0.5,
&      t0, sigma, mode)
  do i = ifrom, ito
    dx(i, ja) = dx(i, ja) - hz_incident * dt_by_dy
  end do
end if
if ((jmin .lt. jb + 1) .and. (jb + 1 .lt. jmax)) then
  yb = y0 + jb * delta_y
  hz_incident = pulse(yb, t - delta_t * 0.5,
&      t0, sigma, mode)
  do i = ifrom, ito
    dx(i, jb + 1) = dx(i, jb + 1) + hz_incident * dt_by_dy
  end do
end if

```

Now, let us get to the most complex correction, namely the one pertaining to H_z . Because:

$$H_z^n(i, j) = H_z^{n-1}(i, j) - \left(\frac{E_y^n(i + 1, j) - E_y^n(i, j)}{\Delta x} - \frac{E_x^n(i, j + 1) - E_x^n(i, j)}{\Delta y} \right) \Delta t$$

the correction in this case should be carried out both on $i = i_a$ and $i = i_b$ as well as on $j = j_a$ and $j = j_b$. But if the incident signal is a plane wave or a plane pulse that propagates in the y direction, then the y component of the incident field must be zero. Consequently, there are not going to be any corrections to H_z on $i = i_a$ and $i = i_b$.

But we are going to have corrections on $j = j_a$ and then on $j = j_b$. Neglecting contribution from E_y we get on $j = j_a$:

$$H_z^n(i, j_a) = H_z^{n-1}(i, j_a) + (E_x^n(i, j_a + 1) - E_x^n(i, j_a)) \Delta t / \Delta y$$

Here $H_z^n(i, j_a)$ and $E_x^n(i, j_a + 1)$ are both *inside* the total field region and only $E_x^n(i, j_a)$ is *outside*. We must *add* this time the incident field to $E_x^n(i, j_a)$ in order for this update to be meaningful. This is how we *inject* the signal into the total field region. The resulting correction is therefore going to be:

$$H_z^n(i, j_a) \leftarrow H_z^n(i, j_a) - E_x^{\text{incident}}(i, j_a, n) \Delta t / \Delta y$$

On the other hand, at $j = j_b$:

$$H_z^n(i, j_b) = H_z^{n-1}(i, j_b) + (E_x^n(i, j_b + 1) - E_x^n(i, j_b)) \Delta t / \Delta y$$

Here both $H_z^n(i, j_b)$ and $E_x^n(i, j_b)$ are inside the total field region, but $E_x^n(i, j_b + 1)$ is outside. So we must again *add* the incident signal to $E_x^n(i, j_b + 1)$ in order for the update to be meaningful. And this is how we *extract* the incident signal from the total field region.

The resulting correction is:

$$H_z^n(i, j_b) \leftarrow H_z^n(i, j_b) + E_x^{\text{incident}}(i, j_b + 1, n) \Delta t / \Delta y$$

These corrections are implemented within subroutine **inject_h** as follows:

```

if ((jmin .lt. ja) .and. (ja .lt. jmax)) then
  ya = y0 + ja * delta_y - delta_y * 0.5
  ex_incident = - pulse(ya, t - delta_t * 0.5,
&      t0, sigma, mode)
  do i = ifrom, ito
    hz(i, ja) = hz(i, ja) - ex_incident * dt_by_dy
  end do
end if
if ((jmin .lt. jb) .and. (jb .lt. jmax)) then
  yb = y0 + jb * delta_y + delta_y * 0.5
  ex_incident = - pulse(yb, t - delta_t * 0.5,
&      t0, sigma, mode)
  do i = ifrom, ito
    hz(i, jb) = hz(i, jb) + ex_incident * dt_by_dy
  end do
end if

```

3.1 Incident Field Models

The incident field is calculated using an analytical formula.

First, let us observe that in the TE mode for an injection of a plane pulse or wave in the y direction we only have H_z and E_x (assume vacuum, so that $\mathbf{D} = \mathbf{E}$). Therefore:

$$\begin{aligned} \partial_t E_x &= \partial_y H_z \\ \partial_t H_z &= \partial_y E_x \end{aligned}$$

Substituting $H_z = f(y - t + t_0)$ (remember that $c = 1$) we find

$$\begin{aligned} \partial_y f(y - t + t_0) &= f' \\ \partial_t f(y - t + t_0) &= -f' \end{aligned}$$

Consequently, we must have that

$$E_x = -H_z$$

which is why we use:

```

        hz_incident = pulse(ya, t - delta_t * 0.5,
&           t0, sigma, mode)

```

in `inject_d` but

```

        ex_incident = - pulse(ya, t - delta_t * 0.5,
&           t0, sigma, mode)

```

in `inject_h`.

As to the shape of the pulse, i.e., function f we provide two models.

Model 1 corresponds to a simple Gaussian pulse:

$$f(y, t) = \exp \left(-\frac{(y - (t - t_0))^2}{2\sigma^2} \right)$$

This is represented in the code of function `pulse` as follows:

```

    if (mode .eq. 1) then
        pulse
&           = exp (- (y - (t - t0)) ** 2 / (2 * sigma ** 2))
    else ...

```

Model 2 corresponds to a slowly ramped harmonic wave:

$$f(y, t) = \frac{1}{2} (\tanh(\alpha(t - t_0 - y)) + 1) \cos(2\pi(y - t)/\sigma)$$

where $\alpha = 0.05$ and is hard-wired into the Fortran code. The ramped harmonic wave is implemented in the code of function `pulse` as follows:

```

    else if (mode .eq. 2) then
        ramp = 0.5 * (tanh(alpha * (t - t0 - y)) + 1.0)
        pulse
&           = ramp * cos(2.0 * pi * (y - t) / sigma)
    end if

```

It is clear from the above that the same parameter, σ , is used either as the half-width of the pulse or as the wavelength of the harmonic wave, depending on the mode. The parameter t_0 represents a delay. It shifts the pulse or the ramp deep into negative y (how deep is determined by t_0) so that the pulse can be turned on *adiabatically* without triggering unphysical effects within the total field region.

4 Medium Distributions

Section 1.3.5, page 13, introduced function `distrib`, which is zero in the absence of the metal medium, one wherever the medium is present, and something in between on the medium boundary. The boundary itself can be made sharp or fuzzy – the program provides various models here.

The interface of function `distrib` is as follows:

```

function distrib (x, y, model, xa, xb, ya, yb)
implicit none
double precision distrib
double precision x, y
integer model
double precision xa, xb, ya, yb

```

The body of function `distrib` is basically a large if statement:

```

    if ((x .lt. xa) .or. (xb .lt. x)
&      .or. (y .lt. ya) .or. (yb .lt. y)) then
        distrib = 0.0D0
    else
        ... blah blah blah ...
    end if

```

This lets us surround the region where metal particles are present with a rectangle and return immediately with `distrib = 0` outside of it.

Inside the rectangle, the following models are implemented:

- 0:** `distrib = 0` everywhere. This model is useful for testing injection, extraction and the propagation of the incident signal itself.
- 1:** `distrib = 1` within a rectangle $[30, 70] \times [40, 70]$, and 0 elsewhere.
- 2:** `distrib = 1` within a circle the centre of which is at (50,50) and radius 15.0, and 0 elsewhere.
- 3:** `distrib = 1` within a circle the centre of which is at (50,50) and radius 14.5, and 0 outside of the circle with the same centre, but radius of 15.5. In between the two radii, the value of `distrib` varies linearly with distance between 0 and 1 according to the formula:

$$-(l - r - \Delta r/2)/\Delta r$$

where l is the distance from the centre of the circle, $r = 15.0$ and $\Delta r = 1$.

- 4:** `distrib` varies inside the rectangle defined by x_a , x_b , y_a and y_b according to the formula:

$$\frac{1}{2} [\tanh(\alpha(r - l)) + 1]$$

where l is the distance from the centre of the circle, (50,50), $r = 15$ and $\alpha = 1$ is a smoothing parameter.

- 5:** Same as model 4 with $\alpha = 0.5$.
- 6:** Same as model 4 with $\alpha = 2.0$.
- 9:** Same as model 4 with $\alpha = 4.0$ – this model was introduced specially for multilevel runs, because $\alpha = 4.0$ makes the gradient very steep.
- 7:** `distrib` is 1 within one of two circles: circle((37.5, 50.5), 7.5) and circle((62.5, 50.5), 7.5), and zero elsewhere.
- 8:** `distrib` is given by the following tanh formula:

$$\frac{1}{2} [\tanh(\alpha(r_1 - l_1)) + 1] + \frac{1}{2} [\tanh(\alpha(r_2 - l_2)) + 1]$$

where $r_1 = r_2 = 7.5$, l_1 is the distance from (37.5, 50.5) and l_2 is the distance from (62.5, 50.5) and $\alpha = 1$.

Both Fortran and Chombo programs were then tested against various models. In these tests I found that the models in which the metal boundary was diffused over a certain width, i.e., models 3, 4, 5, 6, 9 and 8 did not perform well. The thick skin of these models trapped electromagnetic field, which then continued to vibrate within the skin long after the incident pulse has passed through. When the incident signal was a wave the same effect resulted in continued visible ringing within the metal skin. The trick worked quite well in the initial stages of the encounter between the signal and the metal surface and did prevent spiking to some extent.

In multi-level runs, the most effective strategy to prevent the spikes was to resort to models with sharply defined boundaries, i.e., models 1, 2 and 7 and to surround the boundary itself with permanently defined fine mesh – this on top of the dynamically constructed fine mesh that travelled with the incident signal.

5 Fortran Mainlines

The Fortran program, `metal.f`, is a straightforward application of FDTD to a problem of an incident electromagnetic signal scattering on a metal object. Because it is an F77 program, the size of the grid must be hardwired into the code (this wouldn't be the case in F90 and later Fortrans, but there is no GNU F90 yet, so all this development is done with F77). Other parameters though, especially all material and IO specifications, are read from Fortran namelists at the beginning of the program.

After a simple initialization of its basic data structures (all fields are initialized to zero), time-stepping begins and the time stepping loop looks basically as follows:

```
do step = 1, number_of_steps
  t = t - delta_t / 2
  call update_d_pml
  t = t + delta_t
  call inject_d
  call d_to_e
  t = t - delta_t / 2
  call update_h_pml
  t = t + delta_t
  call inject_h
  if (mod (step, stride * image_frequency) .eq. 0) then
    call dump_data
    call extrema
  end if
end do
```

I have removed the very long argument lists from this listing for clarity. An important thing to observe is that time is handled *outside* of update functions. In this case we would be free to do otherwise, but in case of Chombo mainlines time *must* be handled externally – and so Fortran handles it this way too.

Because there is only one level in the Fortran computation, the update functions, both `update_d` and `update_h` are in their PML versions. All problem specific physics, i.e., the type of the material and its distribution within the computational domain is handled by `d_to_e`. This is the only routine a user of the program has to contribute. All the rest, i.e., system time-stepping, PMLs, signal injection and extraction and IO are handled by other subroutines, which are left unchanged.

5.1 Fortran Input

The input to the Fortran program is provided by the means of namelists. There are 6 namelists in the program at present:

chat Specify verbosity.

total_region Specify the geometry of the total field region within the computational domain.

pml_boundary Specify the geometry of the PML boundary.

pulse Specify the character of the incident signal: pulse/wave, what length, what width.

medium Specify the medium and the current model to be used in the computation.

iterations Specify number of iterations, the length of the time step and how often should the images be dumped.

Here is an example of the namelist file:

```
&chat verbose=1 /
&total_region ia=20, ib=80, ja=20, jb=80 /
&pml_boundary no_pml=0, x1=10.0, x2=90.0, y1=10.0, y2=90.0 /
&pulse mode=1, t0=20.0, sigma=10.0 /
&medium no_metal=0, model=7, current_model=1,
        omega_p=0.566, tau=389.604, margin=2.0 /
&iterations number_of_steps=7680, stride=32, image_frequency=2 /
```

In this case the verbosity parameter is set to 1. It can be also zero, in which case the program runs silently, or something larger than 1, in which case the program will chat insanely.

The total field region is specified by providing values for i_a , i_b , j_a and j_b – see section 3.

The width of the PML boundary is specified by providing x_1 , x_2 , y_1 and y_2 . See section 2. It is also possible to switch PMLs off altogether by specifying **no_pml=1**.

The pulse is specified by its **mode** number (i.e., a pulse (1), or a wave (2)), delay, **t0**, and either a pulse half-width or the full wavelength, **sigma**.

Then we have the medium specification. We can switch medium off altogether by defining **no_metal=1** or by defining **model=0**. The meaning of the distribution **model** is described in section 4 and the meaning of the **current_model** is discussed in section 1.3.5. The meaning **omega_p** and **tau** and how they should be evaluated is discussed in section 1.4. Finally, **margin** specifies a margin within the total field region within which **distrib** is going to be set to zero automatically, cf. 4 and the discussion of x_a , x_b , y_a and y_b . The Chombo program lets the user specify these directly, whereas in the Fortran program they are specified by providing **margin**.

Finally we have the specification of the iterations themselves. The total number of steps, the **stride**. The **stride** is a parameter, which divides $\Delta t = \Delta x = 1$ by ... **stride** in order to generate the length of the time step used in the computation. For example, if we want the wave front to move through $\Delta x = 1$ in 16 time steps, then we should set **stride=16**. The last parameter, **image_frequency**, specifies how often do we dump the images. The images are dumped every **stride * image_frequency** time steps.

When the program is run the namelist file must be read into it as follows:

```
$ metal < input
```

6 Chombo Mainlines

The Chombo program is split into several files. The mainlines itself, `metal_Ch.cpp`, is a relatively small file that contains just `main`. The program uses the Chombo `ParmParse` class to collect data from the input file and to make it available to any other C++ class or function within the code that needs it.

The basic data structure within the Chombo program is `level` defined on `metal_Ch.h`:

```
struct level {

    // Tagging specifications

    Real diff_threshold, value_threshold;
    IntVectSet tag_set;

    // Geometry of this level and its distribution amongst the CPUs

    int imin, imax, jmin, jmax, tag_margin;
    Real x0, y0, delta_x, delta_y, xmin, xmax, ymin, ymax;
    Box domain;
    Vector<Box> vector_of_boxes;
    Vector<int> vector_of_processes;

    // Time slices

    Real time_e, time_h, time_e_old, time_h_old;
    Real delta_t;

    // Fields associated with this level.

    LevelData<FArrayBox> D, E, I, S, H, ID, D_old, E_old, I_old, S_old, H_old;

    // ID is used at level 0 only (for PMLs), so there is no need for ID_old

    // Minima and maxima

    Vector<Real> D_max, D_min, E_max, E_min, H_max, H_min;

};
```

All data structures and auxiliary parameters that pertain to a given Chombo/AMR level are defined within `level`. Various vector and scalar fields are objects of class `LevelData`. These objects are really only place-holders and they have to be specially instantiated and sized in order to become real fields (or matrices). Consequently, when a new `level` is created, it is really a very small object with lots of dangling pointers. One has to be aware of it and one has to call `new` or appropriate Chombo functions in order to build the relevant data objects.

Function `main` defines a vector of levels:

```
Vector< level* > levels;
```

which is initially empty, and then calls:

```
build_level(levels);
```

which builds level 0 with all its arrays initialized to zeros. In the process steering input is read from the Chombo input file using `ParmParse`. More about it below.

Now we are ready to enter the iteration loop:

```
for (int label=0, count = 0; count < number_of_steps_0; count++) {

    int time_to_regrid = !((count + 1) % stride_0);
    int time_to_dump_data = !((count + 1) % (stride_0 * image_frequency_0));

    advance_e(levels);
    advance_h(levels);

    if (time_to_regrid) regrid(levels);
    if (time_to_dump_data) {
        analyze_levels(levels);
        dump_data(levels, ++label);
    }
}
```

This looks, I hope, very nice and clean. Of course, there is plenty of dirt swept under the carpet here. What makes it possible is the overloading of most of the functions used by the program, the huge capacity of vector of levels to represent the whole multi-level system, and the ability to pass the remaining parameters around using `ParmParse`. The latter is somewhat similar to the Fortran `common` statement.

The algorithm used to time-step the system and to build higher levels is discussed in detail in [9]. In this part of the project I have merely cleaned the code, niced it up, made it more flexible and more modular, and restructured it for TE and metal. And so, the general idea is that as $\Delta x = \Delta y$ gets halved on moving from level n to level $n + 1$, the corresponding time step Δt gets divided by 3 instead. This way strict leapfrog synchronization between all levels is maintained. The levels are built and advanced recursively.

Where I had separate functions for level 0 and for higher levels in [9], here I merged these functions into a single overloaded function. And so

```
build_level{levels}
```

builds level 0, but

```
build_level{levels, n}
```

builds level n . In both cases, if there are reasons enough to build higher levels, the function will continue calling itself recursively, until all levels have been constructed.

Similarly

```
advance_e{levels}
```

advances level 0, whereas

```
advance_e{levels, n}
```

advances level n . Again, in both cases, if there are higher levels that need to be advanced in synchrony, the function will call itself and its partner, `advance_h`, recursively and in correct order until all levels have been advanced.

Dolly Parton remarked about her own appearance once saying “it takes a lot of money to look this cheap”. I couldn’t agree with her on this point, because to me she will always look wonderful, but I do share the sentiment: “it takes a lot of coding to make the program look this terse.”

6.1 Chombo Input

The chombo input can be divided into separate units similar to Fortran namelists, but at this stage I chose to lump all input parameters into a single list that is divided semantically rather than structurally. And so, we have the following groups of parameters on the `metal_Ch.input` file:

Chat group How much diagnostic output do you want. There are two verbosity parameters here: one is for Chombo, the other one for Fortran. Activating Fortran verbosity should be done with great caution, because the output is going to be extremely copious on account of Fortran subroutines being called in parallel by multiple Chombo/MPI processes.

Level 0 geometry group Specifies the geometry of the level 0 group.

Level 0 iteration group Specifies how level 0 should be iterated, similar to the iterations namelist in the Fortran program.

PML group specifies the width of the PML boundary.

Signal injection group What kind of a signal should be injected, and what should the geometry of the total field region be.

Medium group The geometry of the cut-off region, medium parameters, current model, medium distribution model.

Levels group How many levels, how should they be constructed.

Output style group Activate Gnuplot, HDF5 output. Which fields to output?

Here is an example of the `metal_Ch.input` file:

```
#
# Chat?
#
chombo_verbosity = 0
fortran_verbosity = 0
#
# Level 0 geometry group
#
nx_0 = 100
ny_0 = 100
nbx_0 = 4
x0_0 = 0.0
y0_0 = 0.0
delta_x_0 = 1.0
delta_y_0 = 1.0
#
# Level 0 iteration group
#
time_0 = 0.0
stride_0 = 16
```



```

image_frequency_0 = 2
number_of_steps_0 = 3200
#
# PML group
#
xpml_lo = 10.0
xpml_hi = 90.0
ypml_lo = 10.0
ypml_hi = 90.0
#
# Signal injection group
#
ia = 20
ib = 80
ja = 20
jb = 80
pulse_mode = 1
t0 = 20.0
sigma = 10.0
#
# Medium group
#
omega_p = 0.566
tau = 389.604
distribution_model = 2
current_model = 1
xa = 30.0
xb = 70.0
ya = 30.0
yb = 70.0
#
# Levels group
#
max_number_of_levels = 3
tag_on_diffs = 1
tag_on_values = 1
tag_on_location = 1
tag_margins = 3 5 7
diff_thresholds = 0.045 0.06 0.08
value_thresholds = 0.25 0.5 0.9
xc_tag = 50.0
yc_tag = 50.0
radius_tag = 15.0
half_width_tag = 7.0
#
# Output style group
#
gnuplot_output = 1
hdf5_output = 0
output_Dx = 0
output_Dy = 0
output_Ex = 0
output_Ey = 0
output_Hz = 1

```

```
output_tags = 0
output_boxes = 0
```

In this case the program is going to run silently as both Chombo and Fortran chatting are disabled.

The level 0 grid is going to be 100×100 divided into 4×4 boxes, each box to be assigned to a different CPU if possible. The beginning of the coordinate system for this level is $(0, 0)$ and the grid constants are $\Delta x = 1$ and $\Delta y = 1$.

The iteration begins with $t = 0$. We are going to carry out 3,200 level 0 time steps and 16 time steps will be required for the wave front to cross a distance of $\Delta x = 1$. Images (of all levels) will be dumped every $16 \times 2 = 32$ level 0 time steps.

The PML boundary is going to be 10 cells thick all around the computational domain.

The total field region is defined within the $[20, 80] \times [20, 80]$ rectangle. We are going to inject a pulse of half-width $\sigma = 10$ and with time delay $t_0 = 20$. The medium is going to be silver with $\tau = 389.6$ and $\omega_0 = 0.566$ in natural units for this problem. We are going to have a single cylinder with sharp boundaries (`distribution_model = 2`) and we are going to use the double-current model (`current_model = 1`). The medium cut-off boundary is defined by $x_a = 30, x_b = 70$ and $y_a = 30, y_b = 70$.

We are going to have up to 3 AMR levels throughout the computation. We are going to tag cells for splitting if both values and differences of values in these cells exceed certain limits, specified by `diff_thresholds` and `value_threshold` vectors. What these threshold values have to be must be discovered by experimentation. The tagging is going to be done within progressively narrower regions within the total field region. The `tag_margin` vector tells us that level 1 is going to be 3 cells away from the total field boundary, and level 2 is going to be 5 cells away from the total field boundary.

We are also going to tag cells within a ring of width $2 * \text{half_width_tag}$ and radius `radius_tag`, centered on `(xc_tag, yc_tag)` regardless of field values in them.

In this case we are only going to generate Gnuplot output for H_z . HDF5 output is disabled as are outputs for other fields.

7 Chombo Output

The output of the Chombo program is in the form of data files. These files can be generated either for display with Gnuplot, in which case they are going to be rather voluminous, or for display with Chombovis, in which case the output is in the HDF5 format.

7.1 HDF5 Output

HDF5 files contain all fields specified in the `metal_Ch.input` file, i.e., if we activate, say:

```
output_Ex = 1
output_Ey = 1
output_Hz = 1
```

E_x , E_y and H_z for a given time slice for all levels currently active will be output on a single HDF5 file called, e.g., `fields_027.hdf5`. This file will also contain the gridding information (boxes) for all levels. The output is synchronized on the **E** line and the **H** fields are advanced by half a time step for each level. This means that **E** fields for all levels are dumped at exactly the same time, but **H** fields are dumped for slightly differing times. I would have to time-interpolate these fields in order to achieve the synchronization.



7.2 Gnuplot Output

If Gnuplot output is activated, each field and each level is dumped on a separate file. And so we are going to have files such as:

```
Ex_0_027.dat
Ex_1_027.dat
Ex_2_027.dat
Ey_0_027.dat
Ey_1_027.dat
Ey_2_027.dat
Hz_0_027.dat
Hz_1_027.dat
Hz_2_027.dat
```

Everyone of these files contains a large header with a lot of information. For example a header for `Ex_1_027.dat` may look as follows:

```
# program: metal_Ch, function: write_gnuplot_data
# header:
#   program author: Zdzislaw (Gustav) Meglicki, Indiana University
#   %Id: io_Ch.cpp,v 1.16 2004/11/15 16:02:25 gustav Exp %
#   %Id: metal_Ch.h,v 1.20 2004/11/15 21:11:22 gustav Exp %
#   system kernel: Linux.2.4.26.#1 SMP Wed Apr 21 09:09:56 CDT 2004
#   machine:      i686
#   node:         jlogin1
#   time of dump: Thu Nov 18 11:58:29 2004
#   data for:     Ex
#   level:        1
#   label:        27
#   time_e:       54.000000
#   time_h:       54.010417
#   delta_t:      0.020833
#   delta_t_0:    0.062500
#   xmin:         21.250000
#   xmin_0:       -1.000000
#   xmax:         77.750000
#   xmax_0:       100.000000
#   ymin:         21.250000
#   ymin_0:       -1.000000
#   ymax:         71.750000
#   ymax_0:       100.000000
#   delta_x:      0.500000
#   delta_x_0:    1.000000
#   delta_y:      0.500000
#   delta_y_0:    1.000000
#   data minimum: -1.010016
#   global minimum: -1.010112
#   data maximum: 0.000190
#   global maximum: 0.000190
# data:
```

This information can be used to assemble Gnuplot `*.dat` files for a given field and for a given level into animation. This used to be done by the Chombo program itself, but now I decided to implement

this function as an external shell script. This simplifies the Chombo program itself somewhat. The script is called

```
write_plot_file.sh
```

and it should be invoked as follows:

```
$ write_plot_file.sh Hz 1 > Hz_1.plt
```

The animation can be then displayed by running

```
$ gnuplot Hz_1.plt
```

I continue to support the Gnuplot output mode in the program, because I found it invaluable in debugging and in testing. The Chombovis colour maps look pretty, but they are not as informative as the 3-D surface displays produced with Gnuplot.



8 Tests and Experiments

The program has been run through a fairly large number of tests. These still don't test it against known analytically solvable examples, which is the best test of all, but they test it for consistency against itself, they test Fortran versus Chombo, in case of plane waves and pulses against analytical solutions, and they test various code configurations and models just to check which work best.

8.1 Plane Wave/Pulse propagation

The first test is a plane pulse test: does it propagate through the total field region correctly and does it get correctly injected and removed on the total field region boundary? This is an easy test to run, because if anything goes wrong, there is a visible large signal in the scattered field region. It is also a good test for multi-level runs. If there is anything wrong with the way data is propagated at higher levels, we should see a clear deformation of the wave front.

This test is selected on setting:

```
pulse_mode = 1
t0 = 20.0
sigma = 10.0
distribution_model = 0
max_number_of_levels = 1
gnuplot_output = 1
output_Hz = 1
```

Here we are going to inject a pulse of $\sigma = 10$ into the total field region. There is not going to be any medium within the computational domain other than vacuum (`distribution_model=0`). At this stage we restrict the computation to a single level only.

For a 3-level computation we would change `max_number_of_levels` to 3 and we would add specifications for how to tag cells, e.g.,

```

max_number_of_levels = 3
tag_on_diffs = 0
tag_on_values = 1
tag_on_location = 0
tag_margins = 3 5 7
value_thresholds = 0.5 0.8 0.9

```

Figure 2 illustrates propagation of a pulse in a single-level computation. One can see here that the pulse is cleanly injected, subtracted on the sides and then extracted at the end of its traversal.

On the other hand a close examination of the pulse extraction reveals there a small wave resulting from the difference between numerical and analytical dispersions is generated towards the end of the pulse traversal through the total-field region. This is shown in figure 3

Figure 5 compares the propagation for the pulse between 1- and 3-level computations. Here we are looking at two pulse cross-sections, one for $x \in [45, 50]$ and the other one for $x \in [75, 79]$. The latter corresponds to the slice of the pulse that is very close to the total field boundary, where we don't generate higher level grids. See figure 4.

It is easy to see that in this region the pulses should overlap. On the other hand, in the middle of the domain, i.e., for $x \in [45, 50]$ the 3-level pulse is a little narrower. This is because the numerical dispersion for the 3-level pulse is lower than for the 1-level pulse. The 3-level pulse does not overshoot either, whereas the 1-level pulse has a somewhat higher amplitude than the amplitude of the analytical pulse, which is exactly 1.

There is a price for running computations on 3-levels in this case, apart from using markedly more CPU resources, of course. The price is increased level of noise, which results from these 3 different dispersions and from a rather arbitrary criterion as to which parts of the computational domain should be covered by which levels.

This is best seen when we look at the pulse withdrawal picture through a magnifying glass. This is shown in figure 6, which should be compared with figure 3.

It is possible that we may be able to control or even eliminate this noise by handling data flow on the coarse-fine level boundary much more astutely, for example, by carrying out the refluxing procedure, mentioned in previous papers on a number of occasions (usually with dread). Such a procedure will require a very detailed analytical examination of what exactly happens on the coarse-fine level boundary and how any additional fixes should be implemented.

8.2 Fortran versus Chombo

The second group of tests is to compare data generated by running Fortran mainlines against data generated by running a single-level Chombo job.

Well, this data turns out to be exactly the same for all test situations compared – and these covered various current models, metal distribution models and pulse versus wave.

It is not immediately obvious that it should be so, because even a single level Chombo domain can be partitioned amongst multiple processors, with ghost cell data exchanged between them several times within every time step.

Of course, the general idea is that this should not affect the final results and it is rather rewarding to see that this was indeed the case – at least for this generation of tests which were all run, this should be kept firmly in mind, on a single CPU, even though the data was logically divided into multiple (4×4) plaquettes.

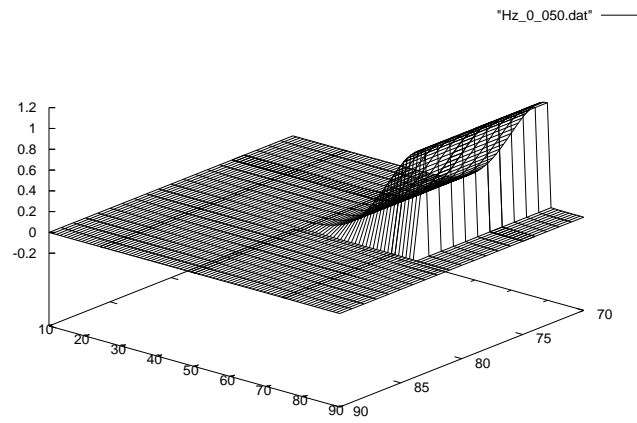
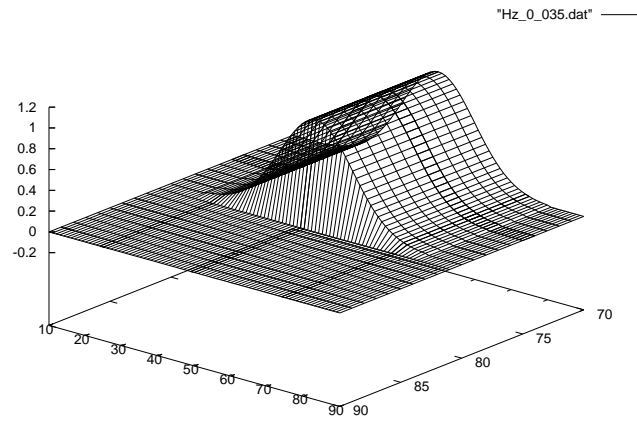
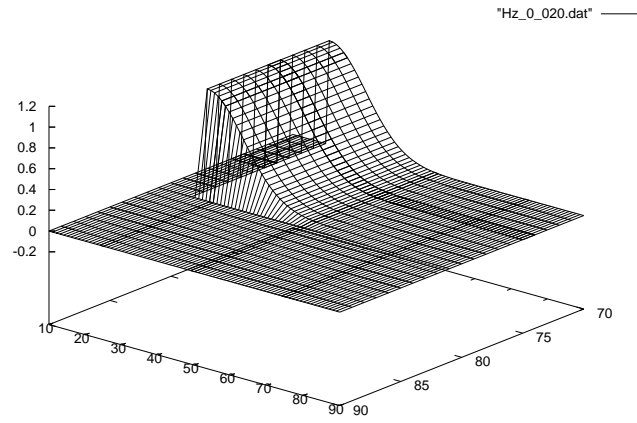


Figure 2: Three snapshots of H_z for a single level pulse propagation test.

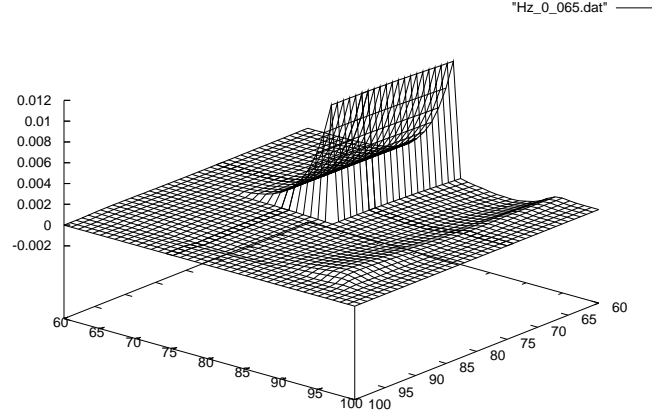


Figure 3: A close examination of the extraction of the pulse shows a small wave resulting from the difference between numerical and analytical dispersion.

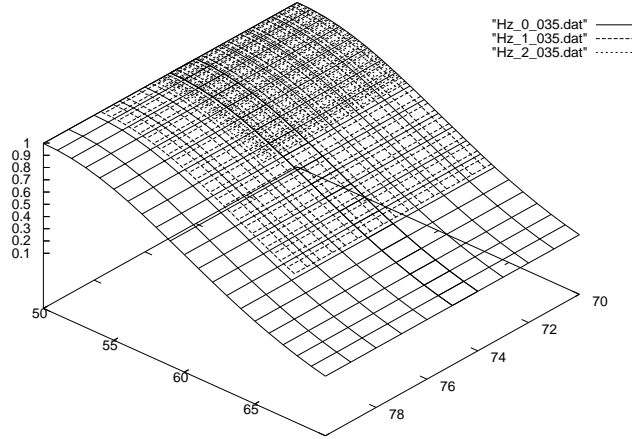


Figure 4: 3-level coverage of H_z at $t_h = 70.03125$ for level 0, $t_h = 70.010417$ for level 1 and $t_h = 70.003472$ for level 2.

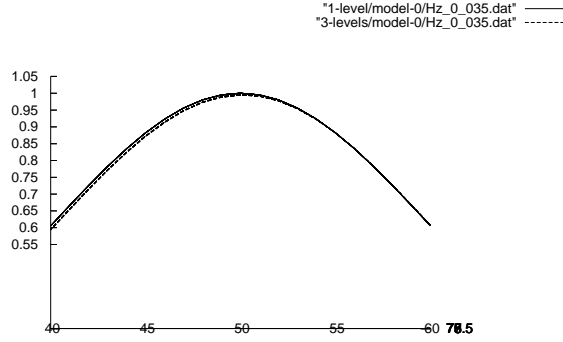
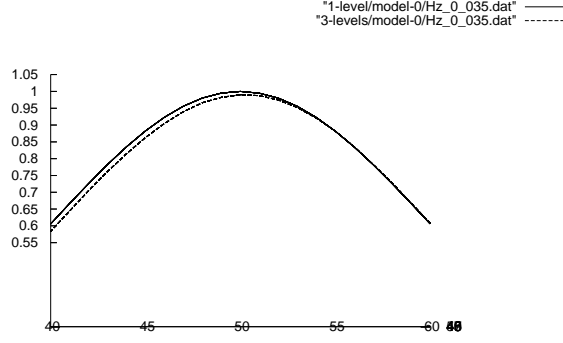


Figure 5: Comparison between 1-level and 3-level runs. The top figure compares the data in the middle of the x -range, i.e., for $x \in [45, 50]$. The bottom figure compares the data on the side of the x -range, i.e., for $x \in [75, 79]$. There is no multi-leveling on the side, because *tag margins* keep us away from tagging cells too close to the total-field boundary.

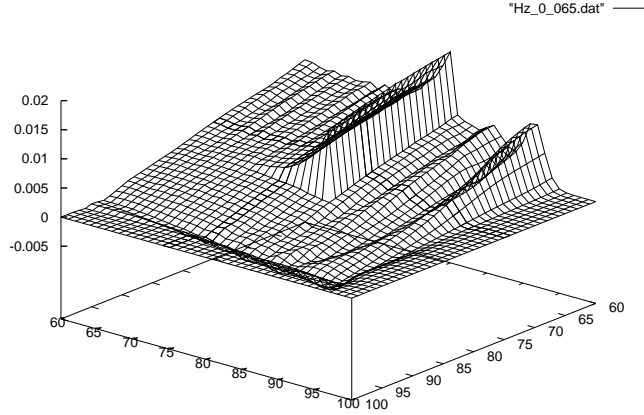


Figure 6: A close examination of the extraction of the pulse for the 3-level run. There is much more noise here compared to figure 3.

8.3 Single Level versus Multi-Level: Quenching of Spikes

The most important question in all of this is, of course, if AMR gives us anything. The following shows that it can have a beneficial effect if used judiciously. But it takes quite a few experiments to arrive at proper threshold levels, and other cell-tagging criteria. Running a multilevel job is quite taxing, because the amount of computation grows dramatically as we refine Chombo grids. Still, because the refined grid is applied only where it is needed, some, perhaps even considerable CPU savings should result compared to a run in which the whole grid would correspond to the finest AMR level. On the other hand if a single grid computation with a very short grid constant could be carried out, e.g., on an ORNL Cray X1 or on an ANL IBM BlueGene, such a computation would probably yield much lower noise level and would be a lot easier to set up: a simple HPF program could be used in this case.

So, let us have a look at the results of a 3-level computation, versus a 1-level one, for a case of a TE pulse scattering on a single round silver cylinder. This corresponds to the metal distribution model 2, cf. section 4. When this problem is attacked on a single-level 100×100 grid snapshots 30 and 35 look as shown in figure 7. On the other hand, when computation for the same situation is carried out on three levels and a fixed high-resolution grid is placed around the periphery of the cylinder, the spikes disappear, as shown in figure 8. We can look at the cylinder boundary in as much magnification as this 3-level computation affords. Figure 9 shows H_z at the edge of the cylinder obtained from level 2 data. The saw-tooth approximation of the curve is clearly seen, but... there are no spikes!

It is instructive to look also at the \mathbf{E} field at various levels. And so, figure 10 shows fields E_x and E_y for snapshot number 30 at levels 0 and 2. Here it is easy to see why the computation is so difficult and why being able to increase the resolution at least in certain locations helps. We have to deal with very steep gradients of \mathbf{E} on the cylinder boundary. Figure 10 shows that there is an unhealthy fluctuation of \mathbf{E} right at the boundary visible in level 2 data. Luckily the amplitude of these fluctuations is not great and fluctuations in E_x and in E_y cancel so that they don't carry to H_z . Nevertheless this picture does suggest that we may consider adding a yet another level of computation to the system.

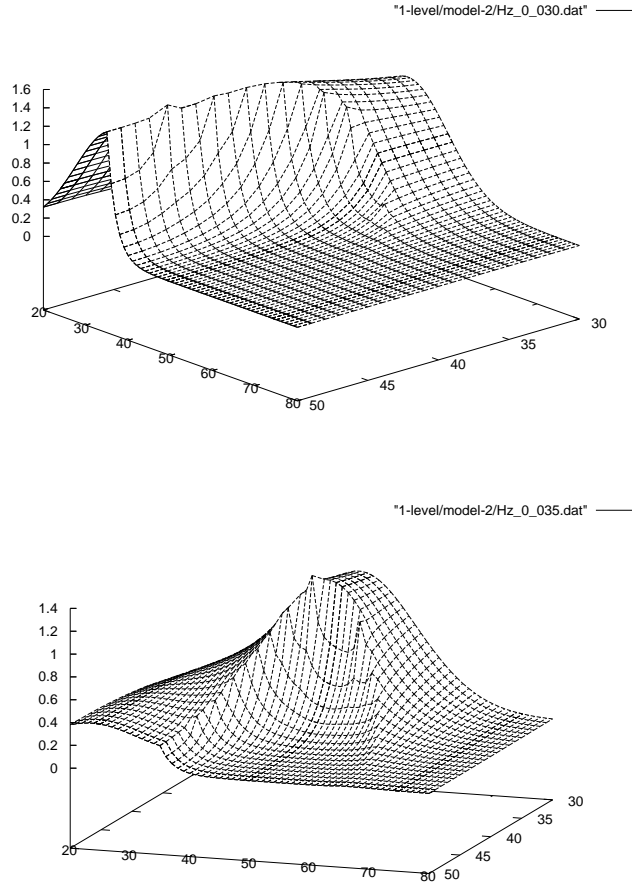


Figure 7: H_z spikes form on the cylinder boundary when the pulse passes through it. This occurs for one-level simulations.

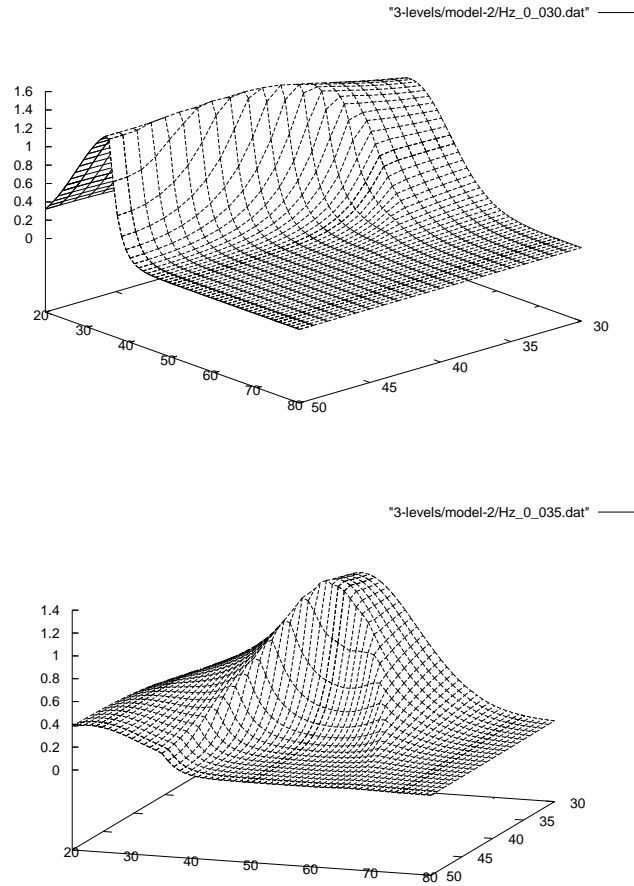


Figure 8: H_z spikes disappear from the cylinder boundary when simulation is carried out on 3 levels with high resolution mesh covering the boundary of the cylinder at all times.

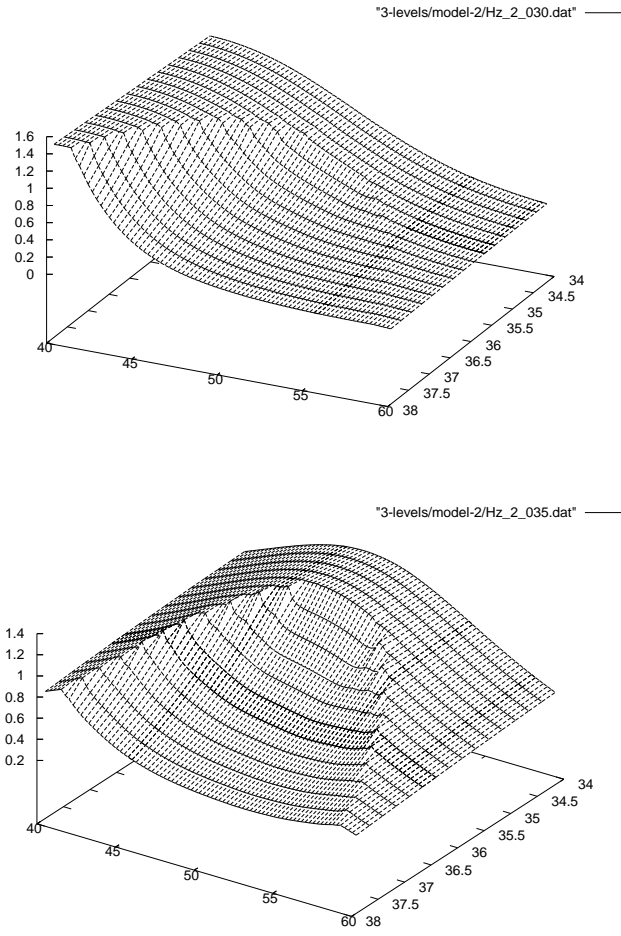


Figure 9: A magnified level-2 image of the cylinder boundary for snapshots number 30 and 35.

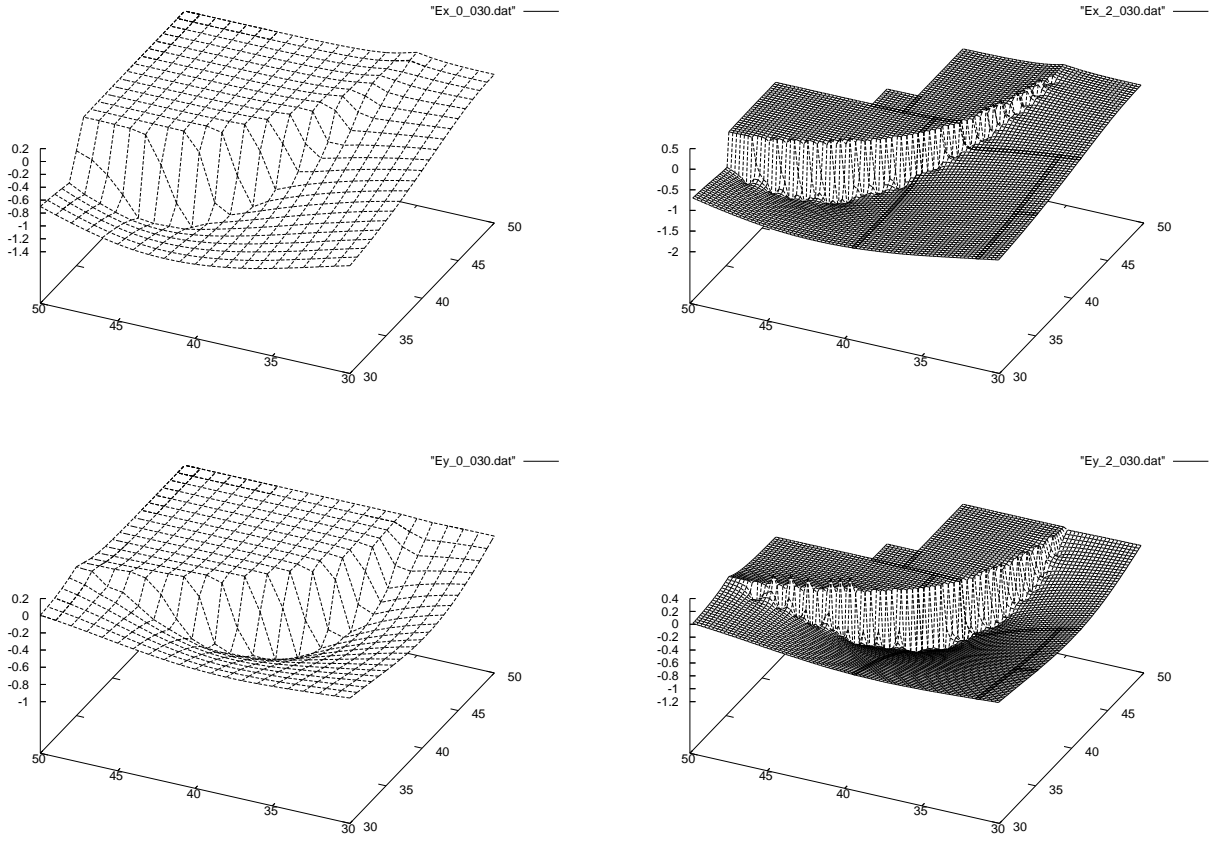


Figure 10: Fields E_x and E_y at two levels of resolution (0 and 2), snapshot number 30.

9 What Next

9.1 Parallelization

In order for this code to be usable, it needs to run on a multiprocessor. Right now simulations with 100×100 resolution at level 0 through 400×400 resolution at level 2 are possible, but we would really like to go to something like $10,000 \times 10,000$ at level 2. A system like this cannot be tackled on a single PC, but it can be attacked on a farm.

The code is parallel in principle already. For example, all constructs such as:

```
for (data_iterator_n.reset(); data_iterator_n.ok();
    ++data_iterator_n)
    update_d_(
        &(D_n[data_iterator_n()].loVect()[0]),    // imin
        &(D_n[data_iterator_n()].hiVect()[0]),    // imax
        &(D_n[data_iterator_n()].loVect()[1]),    // jmin
        &(D_n[data_iterator_n()].hiVect()[1]),    // jmax
        D_n[data_iterator_n()].dataPtr(0),        // dx
        D_n[data_iterator_n()].dataPtr(1),        // dy
        H_n[data_iterator_n()].dataPtr(0),        // hz
        &delta_x_n,                               // delta_x
        &delta_y_n,                               // delta_y
        &time_e_n,                                // t
        &delta_t_n,                               // delta_t
        &fortran_verbose                          // verbose
    );
D_n.exchange(two_slots);
time_e_n = time_e_n + delta_t_n;
E_fill_boundary.fillInterp(D_n, D_coarse_old, D_coarse, 1.0/3.0, 0, 0, 2);
```

are parallel. Here Chombo is going to execute `update_d_` on plaquettes that live on separate CPUs, if such have been configured into the system. The call `D_n.exchange(two_slots)` will exchange ghost nodes data between CPUs. Finally `E_fill_boundary.fillInterp` will carry out interpolation of fine-level boundary data from a coarse grid to the fine one also in parallel on various CPUs, each looking after its own plaquettes. Chombo takes care of moving data between CPUs in this case, since the coarse grid processor layout may well be different from the fine grid processor layout.

On the other hand, constructs such as scanning through all boxes in order to find minimum and maximum, as implemented in function `analyze_levels`, defined on `levels_Ch.cpp`, are *not* parallel, even though Chombo will attempt to execute them in parallel on a multiprocessor. The reason for the break in parallelism is that the current function assumes that all CPUs can see data from all other CPUs, in particular, the min/max data. This is not the case. Each Chombo CPU can only see its own data. If a data exchange needs to be carried out other than the exchange method for filling ghost nodes, an explicit MPI gather or scatter operation must be invoked. Chombo provides auxiliary utilities for doing this, and what's needed at this stage is to comb through the existing code, locate places, where explicit scatter or gather must be invoked and modify the code as needed.

There are only a few such places and the ones that I am aware of are linked to Gnuplot graphics, which the parallel version is going to disable anyway.

But before we can run this program in parallel, we'll have to rebuild Chombo libraries and link them with MPI, MPI-IO, HDF5 and HDF5/MPI. Then we'll have to link the Chombo program with Chombo libraries so constructed. MPI-IO itself must be configured so that it supports PVFS.



9.2 Gnuplot Graphics

Gnuplot graphics proved enormously useful in this project. To be able to view a 3-d surface representing a 2-d field provides a very precise and quick way of checking the computation. One can easily superimpose several fields on a single display and debug even quite complex problems. A single spike or instability can be easily located. Colour maps provided by ChomboVis for 2D Chombo data are not this useful.

Yet Gnuplot cannot be used in the MPI mode – well, not easily and certainly not on very large data sets. So we'll have to switch to HDF5 data dumps and to ChomboVis for visualization of the data. After about 2 weeks of struggle, I managed to get ChomboVis going under Cygwin. It should be much easier under Linux. A Linux version of ChomboVis is running on MCS workstations.

Two things could be done in order to display HDF5 data as 3-d surfaces. We could write an auxiliary program that would extract data from HDF5 files and would write it out as Gnuplot data files. Another option would be to make Rocketeer read HDF5 data dumped by Chombo applications. This would give us another tool, apart from ChomboVis, for data analysis.

9.3 Production 2D Code

This version of the code has been revised extensively in order to make it a production code. The logic of the code is much cleaner, as is movement and distribution of data within the code. Nearly all parameters can be passed to the run through the input file.

The next version of the code will extract `d_to_e` from the code and reformulate it in Chombo Fortran. Chombo Fortran is somewhat reminiscent of F90. It is a simple preprocessor that takes an input file and generates required Chombo headers and interfaces automatically, while compiling it to standard F77 at the same time. The resulting system should be then quite easy to use. The user will be burdened basically with writing `d_to_e` only, and then plugging it into the multi-level Chombo program for compilation and parallel execution.

Once we have reached this stage, the code will be frozen and configured into the ANL BitKeeper for future maintenance and revisions. In this form it will be also used for further code correctness tests.

A simple user manual, apart from the code annotation document, will be developed.

9.4 3D code

After the current 2D code is completed, the next stage will be to develop a similar 3D code. I have, I think, a well defined algorithm for electrodynamic refluxing in the 3D case, that derives directly from the 3D version of Maxwell equations in the integral form. So this feature can be added and tested. But 3D codes in general are very hard to debug, because of the amount of data they generate. Development of the 3D code will require access to high-bandwidth visualization facilities.

References

- [1] Neil W. Ashcroft, N. David Mermin and David Mermin, “Solid State Physics”, Brooks Cole, 1976
- [2] Chombo: see <http://seesar.lbl.gov/anag/chombo/>
- [3] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini, B. Van Straalen, “Chombo Software Package for AMR Applications Design Document”, Applied Numerical Algorithms Group, NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA, September 12, 2003
- [4] Richard P. Feynman, Robert B. Leighton, Matthew Sands, “The Feynman Lectures on Physics”, Addison-Wesley, Fourteenth Printing, 1981.
- [5] Gnuplot: see <http://www.gnuplot.info/>
- [6] HDF5: see <http://hdf.ncsa.uiuc.edu/HDF5/>
- [7] Zdzislaw Meglicki, “Computational Electrodynamics with Chombo, Part I”,
<http://www-unix.mcs.anl.gov/~meglicki/AMR/index.html>
- [8] Zdzislaw Meglicki, “PML and Staggered Grids in Chombo”,
<http://www-unix.mcs.anl.gov/~meglicki/PML/index.html>
- [9] Zdzislaw Meglicki, “Computational Electrodynamics with Chombo, Part II”,
<http://www-unix.mcs.anl.gov/~meglicki/Cylinder-AMR-3L/index.html>
- [10] Michael Nielsen, “Lectures on Metals and Superconductors”,
<http://www.qinfo.org/people/nielsen/blog/archive/000037.html>
- [11] Z. S. Sacks, D. M. Kingsland, R. Lee, J. F. Lee, “A perfectly matched anisotropic absorber for use as an absorbing boundary condition”, IEEE Transactions on Antennas and Propagation, vol. 43, December 1995, pp. 1460-1463
- [12] Dennis M. Sullivan, “An unsplit step 3-D PML for use with the FDTD method”, IEEE Microwave and Guided Wave Letters, vol. 7, July 1997, pp. 184-186.
- [13] Dennis M. Sullivan, “Electromagnetic Simulation Using the FDTD Method”, IEEE Press Series on RF and Microwave Technology, IEEE Press, New York, 2000, ISBN 0-7803-4747-1
- [14] Allen Taflov, Susan C. Hagness, “Computational Electrodynamics”, Second Edition, Artech House, Boston, London, 2000, ISBN 1-58053-076-1
- [15] Hai Tao, “CMPE 163: Multimedia Processing and Applications”, Lecture Notes, Spring 2003,
<http://www.soe.ucsc.edu/classes/cmpe163/Spring03/>

Index

- 2D production code, 47
- 3D code, 47
- Ashcroft, Neil W., 3
- Avogadro number, 3
- CFL stability criterion, 16
- Chombo, 30
 - AMR levels, 30
 - plane wave propagation, 36
 - AMR time step, 31
 - Gnuplot output, 35, 47
 - in parallel runs, 47
 - write_plot_file.sh, 36
 - HDF5, 34
 - parallelism
 - explicit, 46
 - implicit, 46
 - ParmParse, 30
 - input, 32
 - similarity to common, 31
- Drude model, 3
 - Maxwell equations, 4
 - in frequency domain, 5
 - in natural units, 7, 16
 - in time domain, 7
 - validity, 3
- Feynman, Richard, 3, 5
- Fortran
 - namelist, 28, 29
- Mermin, David, 3
- Mermin, N. David, 3
- Nielsen, Michael, 3
- parallel execution, 46
- Parton, Dolly, 32
- solving $D(E)$, 8
 - auxiliary differential equation, 12
 - code implementation, 13
 - double current method, 9
 - single current method, 9
 - Z-transform, 10
- TE equations
 - in frequency domain, 18
 - in time domain, 19
 - discretization, 19, 37
 - medium distribution, 26
 - multi-level execution, 28
 - PML ABCs, 18
 - code implementation, 20
 - total/scattered field, 21
 - code implementation, 23–25
 - incident field models, 25